The Open Source Reader

[1] The Cathedral and the Bazaar
[2] Homesteading the Noosphere
[3] The Magic Cauldron
By Eric S. Raymond

[4] Halloween 1
[5] Halloween 2
By Microsoft Staff, commented by Eric S. Raymond

[6] The GNU Manifesto
[7] GNU General Public License
By Richard Stallman

[8] The Cluetrain Manifesto
By Christopher Locke, et al.

[9] Free Software Leaders Stand Together
By Bruce Perens, et al.

[10] Suggested Web Sites

Compilation By:
Felipe Csaszar
www.csaszar.org

Version 1.2, 2001-11-22 (*)
(*) Let's join forces and represent the date with an unambiguous format. Use ISO 8601 (YYYY-MM-DD).

Introduction

Open Source, to free the source code of an application, is a radical concept. At first sight it seems anti economic. Because people are "rational" (in an economic sense), it should not be something very successful or extended.

Reality shows exactly the opposite, thousands of high quality programs are free. Each year we see more and more software joining the free software pool. The quality of several open source systems is widely recognized and in many cases outperforms its commercial counterparts.

Notorious examples of high quality open source applications are the Linux kernel, the GNU compiler suite, Apache, Sendmail, KDE, Gnome and StarOffice. Together, these programs makes Linux a leader in the sever arena, and make a step towards being a prominent player in the desktop, handheld and appliance markets.

To fully understand the Open Source philosophy, origins and applications, requires reading some key material.

The motivation for making this compilation was to put into one package everything is needed to understand the Open Source Software Movement. I developed an earlier version of this compilation for myself, to have food for thought while on a short vacation. Lately this compilation has been beta tested by two classes of Computer Science students of the Universidad de Chile and Universidad Católica, during a course on Internet Entrepreneurship I taught.

This file is a compact PDF, which allows for easy on-screen reading, printing and emailing. I also made an effort to typeset everything so that it is readable. It has a consistent layout across sections, a minimum number of pages and a Table of Contents.

Please feel free to expand this virus.... make other people understand what is Open Source Software is and how best to use it.

Enjoy your reading,

Felipe Csaszar
http://www.csaszar.org

The Open Source Reader is also “open source”

The Open Source Reader is also “open source”. If you want to edit it, or add new material, contact me (felipe dot csaszar at csaszar dot org) and ask me to send you the sources (a StarOffice document). I would also appreciate your feedback to improve future editions of the Open Source Reader.

Acknowledgments

Thanks to:

- **David Wheeler** [dwheeler at dwheeler dot com] for his several ideas, including the addition of an Index.
- **Stephen Toothman** [stoothman at yahoo dot com] for the proofreading.
- **Greg Willden** [gregory dot willden at swri dot org] for the layout comments.
- **Jonathan Byron** [geodigest at telocity dot com] for the mirroring (http://64.128.176.121:8080/geodigest/byron/opensourcereader/index.html) and the content comments.
- **Art Wildman** [wildman at mediaone dot net] for being the first suggesting “Free Software Leaders Stand Together”.
- **Barbara** [BT3650 at aol dot com], **Randal Leavitt** [randal dot leavitt at home dot com], **Sagar** [yosagar at yahoo dot co dot uk] and **Hyy** [hlovek at 163 dot net] for the good vibrations.
- And to the thousands of people who has downloaded this document.

Table of Contents

The Cathedral and the Bazaar

1. The Cathedral and the Bazaar

Linux is subversive. Who would have thought even five years ago (1991) that a world-class operating system could coalesce as if by magic out of part-time hacking by several thousand developers scattered all over the planet, connected only by the tenuous strands of the Internet?

Certainly not I. By the time Linux swam onto my radar screen in early 1993, I had already been involved in Unix and open-source development for ten years. I was one of the first GNU contributors in the mid-1980s. I had released a good deal of open-source software onto the net, developing or co-developing several programs (nethack, Emacs's VC and GUD modes, xlife, and others) that are still in wide use today. I thought I knew how it was done.

Linux overturned much of what I thought I knew. I had been preaching the Unix gospel of small tools, rapid prototyping and evolutionary programming for years. But I also believed there was a certain critical complexity above which a more centralized, a priori approach was required. I believed that the most important software (operating systems and really large tools like the Emacs programming editor) needed to be built like cathedrals, carefully crafted by individual wizards or small bands of mages working in splendid isolation, with no beta to be released before its time.

Linus Torvalds's style of development - release early and often, delegate everything you can, be open to the point of promiscuity - came as a surprise. No quiet, reverent cathedral-building here -- rather, the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches (aptly symbolized by the Linux archive sites, who'd take submissions from **anyone**) out of which a coherent and stable system could seemingly emerge only by a succession of miracles.

The fact that this bazaar style seemed to work, and work well, came as a distinct shock. As I learned my way around, I worked hard not just at individual projects, but also at trying to understand why the Linux world not only didn't fly apart in confusion but seemed to go from strength to strength at a speed barely imaginable to cathedral-builders.

By mid-1996 I thought I was beginning to understand. Chance handed me a perfect way to test my theory, in the form of an open-source project that I could consciously try to run in the bazaar style. So I did -- and it was a significant success.

This is the story of that project. I'll use it to propose some aphorisms about effective open-source development. Not all of these are things I first learned in the Linux world, but we'll see how the Linux world gives them particular point. If I'm correct, they'll help you understand exactly what it is that makes the Linux community such a fountain of good software -- and, perhaps, they will help you become more productive yourself.

2. The Mail Must Get Through

Since 1993 I'd been running the technical side of a small free-access Internet service provider called Chester County InterLink (CCIL) in West Chester, Pennsylvania. I co-founded CCIL and wrote our unique multiuser bulletin-board software -- you can check it out by telnetting to locke.ccil.org. Today it supports almost three thousand users on thirty lines. The job allowed me 24-hour-a-day access to the net through CCIL's 56K line -- in fact, the job practically demanded it!

I had gotten quite used to instant Internet email. I found having to periodically telnet over to locke to check my mail annoying. What I wanted was for my mail to be delivered on snark (my home system) so that I would be notified when it arrived and could handle it using all my local tools.

The Internet's native mail forwarding protocol, SMTP (Simple Mail Transfer Protocol), wouldn't suit, because it works best when machines are connected full-time, while my personal machine isn't always on the net, and doesn't have a static IP address. What I needed was a program that would reach out over my intermittent dialup connection and pull across my mail to be delivered locally. I knew such things existed, and that most of them used a simple application protocol called POP (Post Office Protocol). POP is now widely supported by most common mail clients, but at the time, it wasn't built-in to the mail reader I was using.

I needed a POP3 client. So I went out on the net and found one. Actually, I found three or four. I used one of them for a while, but it was missing what seemed an obvious feature, the ability to hack the addresses on fetched mail so replies would work properly.

The problem was this: suppose someone named `joe' on locke sent me mail. If I fetched the mail to snark and then tried to reply to it, my mailer would cheerfully try to ship it to a nonexistent `joe' on snark. Hand-editing reply addresses to tack on `@ccil.org' quickly got to be a serious pain.

This was clearly something the computer ought to be doing for me. But none of the existing POP clients knew how! And this brings us to the first lesson:

1. Every good work of software starts by scratching a developer's personal itch.

Perhaps this should have been obvious (it's long been proverbial that ``Necessity is the mother of invention'') but too often software developers spend their days grinding away for pay at programs they neither need nor love. But not in the Linux world – which may explain why the average quality of software originated in the Linux community is so high.

So, did I immediately launch into a furious whirl of coding up a brand-new POP3 client to compete with the existing ones? Not on your life! I looked carefully at the POP utilities I had in hand, asking myself ``which one is closest to what I want?'' Because

2. Good programmers know what to write. Great ones know what to rewrite (and reuse).

While I don't claim to be a great programmer, I try to imitate one. An important trait of the great ones is constructive laziness. They know that you get an A not for effort but for results, and that it's almost always easier to start from a good partial solution than from nothing at all.

Linus Torvalds, for example, didn't actually try to write Linux from scratch. Instead, he started by reusing code and ideas from Minix, a tiny Unix-like operating system for PC clones. Eventually all the Minix code went away or was completely rewritten -- but while it was there, it provided scaffolding for the infant that would eventually become Linux.

In the same spirit, I went looking for an existing POP utility that was reasonably well coded, to use as a development base.

The source-sharing tradition of the Unix world has always been friendly to code reuse (this is why the GNU project chose Unix as a base OS, in spite of serious reservations about the OS itself). The Linux world has taken this tradition nearly to its technological limit; it has terabytes of open sources generally available. So spending time looking for some else's almost-good-enough is more likely to give you good results in the Linux world than anywhere else.

And it did for me. With those I'd found earlier, my second search made up a total of nine candidates – fetchpop, PopTart, getmail, gwpop, pimp, pop-perl, popc, popmail and upop. The one I first settled on was `fetchpop' by Seung-Hong Oh. I put my header-rewrite feature in it, and made various other improvements which the author accepted into his 1.9 release.

A few weeks later, though, I stumbled across the code for `popclient' by Carl Harris, and found I had a problem. Though fetchpop had some good original ideas in it (such as its background-daemon mode), it could only handle POP3 and was rather amateurishly coded (Seung-Hong was at that time a bright but inexperienced programmer, and both traits showed). Carl's code was better, quite professional and solid, but his program lacked several important and rather tricky-to-implement fetchpop features (including those I'd coded myself).

Stay or switch? If I switched, I'd be throwing away the coding I'd already done in exchange for a better development base.

A practical motive to switch was the presence of multiple-protocol support. POP3 is the most commonly used of the post-office server protocols, but not the only one. Fetchpop and the other competition didn't do POP2, RPOP, or APOP, and I was already having vague thoughts of perhaps adding IMAP (Internet Message Access Protocol, the most recently designed and most powerful post-office protocol) just for fun.

But I had a more theoretical reason to think switching might be as good an idea as well, something I learned long before Linux.

3. ``Plan to throw one away; you will, anyhow.'' (Fred Brooks, ``The Mythical Man-Month'', Chapter 11)

Or, to put it another way, you often don't really understand the problem until after the first time you implement a solution. The second time, maybe you know enough to do it right. So if you want to get it right, be ready to start over **at least** once [JB] .

Well (I told myself) the changes to fetchpop had been my first try. So I switched.

After I sent my first set of popclient patches to Carl Harris on 25 June 1996, I found out that he had basically lost interest in popclient some time before. The code was a bit dusty, with minor bugs hanging out. I had many changes to make, and we quickly agreed that the logical thing for me to do was take over the program.

Without my actually noticing, the project had escalated. No longer was I just contemplating minor patches to an existing POP client. I took on maintaining an entire one, and there were ideas bubbling in my head that I knew would probably lead to major changes.

In a software culture that encourages code-sharing, this is a natural way for a project to evolve. I was acting out this principle:

4. If you have the right attitude, interesting problems will find you.

But Carl Harris's attitude was even more important. He understood that

5. When you lose interest in a program, your last duty to it is to hand it off to a competent successor.

Without ever having to discuss it, Carl and I knew we had a common goal of having the best solution out there. The only question for either of us was whether I could establish that I was a safe pair of hands. Once I did that, he acted with grace and dispatch. I hope I will do as well when it comes my turn.

3. The Importance of Having Users

And so I inherited popclient. Just as importantly, I inherited popclient's user base. Users are wonderful things to have, and not just because they demonstrate that you're serving a need, that you've done something right. Properly cultivated, they can become co-developers.

Another strength of the Unix tradition, one that Linux pushes to a happy extreme, is that a lot of users are hackers too. Because source code is available, they can be **effective** hackers. This can be tremendously useful for shortening debugging time. Given a bit of encouragement, your users will diagnose problems, suggest fixes, and help improve the code far more quickly than you could unaided.

6. Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging.

The power of this effect is easy to underestimate. In fact, pretty well all of us in the open-source world drastically underestimated how well it would scale up with number of users and against system complexity, until Linus Torvalds showed us differently.

In fact, I think Linus's cleverest and most consequential hack was not the construction of the Linux kernel itself, but rather his invention of the Linux development model. When I expressed this opinion in his presence once, he smiled and quietly repeated something he has often said: ``I'm basically a very lazy person who likes to get credit for things other people actually do.'' Lazy like a fox. Or, as Robert Heinlein famously wrote of one of his characters, too lazy to fail.

In retrospect, one precedent for the methods and success of Linux can be seen in the development of the GNU Emacs Lisp library and Lisp code archives. In contrast to the cathedral-building style of the Emacs C core and most other GNU tools, the evolution of the Lisp code pool was fluid and very user-driven. Ideas and prototype modes were often rewritten three or four times before reaching a stable final form. And loosely-coupled collaborations enabled by the Internet, a la Linux, were frequent.

Indeed, my own most successful single hack previous to fetchmail was probably Emacs VC (version control) mode, a Linux-like collaboration by email with three other people, only one of whom (Richard Stallman, the author of Emacs and founder of the Free Software Foundation) I have met to this day. It was a front-end for SCCS, RCS and later CVS from within Emacs that offered ``one-touch'' version control operations. It evolved from a tiny, crude sccs.el mode somebody else had written. And the development of VC succeeded because, unlike Emacs itself, Emacs Lisp code could go through release/test/improve generations very quickly.

4. Release Early, Release Often

Early and frequent releases are a critical part of the Linux development model. Most developers (including me) used to believe this was bad policy for larger than trivial projects,

because early versions are almost by definition buggy versions and you don't want to wear out the patience of your users.

This belief reinforced the general commitment to a cathedral-building style of development. If the overriding objective was for users to see as few bugs as possible, why then you'd only release a version every six months (or less often), and work like a dog on debugging between releases. The Emacs C core was developed this way. The Lisp library, in effect, was not -- because there were active Lisp archives outside the FSF's control, where you could go to find new and development code versions independently of Emacs's release cycle [QR].

The most important of these, the Ohio State elisp archive, anticipated the spirit and many of the features of today's big Linux archives. But few of us really thought very hard about what we were doing, or about what the very existence of that archive suggested about problems in the FSF's cathedral-building development model. I made one serious attempt around 1992 to get a lot of the Ohio code formally merged into the official Emacs Lisp library. I ran into political trouble and was largely unsuccessful.

But by a year later, as Linux became widely visible, it was clear that something different and much healthier was going on there. Linus's open development policy was the very opposite of cathedral-building. Linux's Internet archives were burgeoning, multiple distributions were being floated. And all of this was driven by an unheard-of frequency of core system releases.

Linus was treating his users as co-developers in the most effective possible way:

7. Release early. Release often. And listen to your customers.

Linus's innovation wasn't so much in doing quick-turnaround releases incorporating lots of user feedback (something like this had been Unix-world tradition for a long time), but in scaling it up to a level of intensity that matched the complexity of what he was developing. In those early times (around 1991) it wasn't unknown for him to release a new kernel more than once a **day!** Because he cultivated his base of co-developers and leveraged the Internet for collaboration harder than anyone else, this worked.

But **how** did it work? And was it something I could duplicate, or did it rely on some unique genius of Linus Torvalds?

I didn't think so. Granted, Linus is a damn fine hacker. How many of us could engineer an entire production-quality operating system kernel from scratch?. But Linux didn't represent any awesome conceptual leap forward. Linus is not (or at least, not yet) an innovative genius of design in the way that, say, Richard Stallman or James Gosling (of NeWS and Java) are. Rather, Linus seems to me to be a genius of engineering, with a sixth sense for avoiding bugs and development dead-ends and a true knack for finding the minimum-effort path from point A to point B. Indeed, the whole design of Linux breathes this quality and mirrors Linus's essentially conservative and simplifying design approach.

So, if rapid releases and leveraging the Internet medium to the hilt were not accidents but integral parts of Linus's engineering-genius insight into the minimum-effort path, what was he maximizing? What was he cranking out of the machinery?

Put that way, the question answers itself. Linus was keeping his hacker/users constantly stimulated and rewarded -- stimulated by the prospect of having an ego-satisfying piece of the action, rewarded by the sight of constant (even **daily**) improvement in their work.

Linus was directly aiming to maximize the number of person-hours thrown at debugging and development, even at the possible cost of instability in the code and user-base burnout if any serious bug proved intractable. Linus was behaving as though he believed something like this:

8. Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone.

Or, less formally, ``Given enough eyeballs, all bugs are shallow.'' I dub this: ``Linus's Law''.

My original formulation was that every problem ``will be transparent to somebody''. Linus demurred that the person who understands and fixes the problem is not necessarily or even usually the person who first characterizes it. ``Somebody finds the problem,'' he says, ``and somebody **else** understands it. And I'll go on record as saying that finding it is the bigger challenge.'' But the point is that both things tend to happen rapidly.

Here, I think, is the core difference underlying the cathedral-builder and bazaar styles. In the cathedral-builder view of programming, bugs and development problems are tricky, insidious, deep phenomena. It takes months of scrutiny by a dedicated few to develop confidence that you've winkled them all out. Thus the long release intervals, and the inevitable disappointment when long-awaited releases are not perfect.

In the bazaar view, on the other hand, you assume that bugs are generally shallow phenomena -- or, at least, that they turn shallow pretty quickly when exposed to a thousand eager co-developers pounding on every single new release. Accordingly you release often in order to get more corrections, and as a beneficial side effect you have less to lose if an occasional botch gets out the door.

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

Maybe it shouldn't have been such a surprise, at that. Sociologists years ago discovered that the averaged opinion of a mass of equally expert (or equally ignorant) observers is quite a bit more reliable a predictor than that of a single randomly-chosen one of the observers. They called this the ``Delphi effect''. It appears that what Linus has shown is that this applies even to debugging an operating system -- that the Delphi effect can tame development complexity even at the complexity level of an OS kernel.

One special feature of the Linux situation that clearly helps along the Delphi effect is the fact that the contributors for any given project are self-selected. An early respondent pointed out that contributions are received not from a random sample, but from people who are interested enough to use the software, learn about how it works, attempt to find solutions to problems they encounter, and actually produce an apparently reasonable fix. Anyone who passes all these filters is highly likely to have something useful to contribute.

I am indebted to my friend Jeff Dutky <dutky@wam.umd.edu> for pointing out that Linus's Law can be rephrased as ``Debugging is parallelizable''. Jeff observes that although debugging requires debuggers to communicate with some coordinating developer, it doesn't require significant coordination between debuggers. Thus

it doesn't fall prey to the same quadratic complexity and management costs that make adding developers problematic.

In practice, the theoretical loss of efficiency due to duplication of work by debuggers almost never seems to be an issue in the Linux world. One effect of a ``release early and often policy'' is to minimize such duplication by propagating fed-back fixes quickly [JH] .

Brooks (the author of ``The Mythical Man-Month'') even made an off-hand observation related to Jeff's: ``The total cost of maintaining a widely used program is typically 40 percent or more of the cost of developing it. Surprisingly this cost is strongly affected by the number of users. **More users find more bugs** .'' (my emphasis).

More users find more bugs because adding more users adds more different ways of stressing the program. This effect is amplified when the users are co-developers. Each one approaches the task of bug characterization with a slightly different perceptual set and analytical toolkit, a different angle on the problem. The ``Delphi effect'' seems to work precisely because of this variation. In the specific context of debugging, the variation also tends to reduce duplication of effort.

So adding more beta-testers may not reduce the complexity of the current ``deepest'' bug from the **developer's** point of view, but it increases the probability that someone's toolkit will be matched to the problem in such a way that the bug is shallow **to that person**.

Linus coppers his bets, too. In case there **are** serious bugs, Linux kernel version are numbered in such a way that potential users can make a choice either to run the last version designated ``stable'' or to ride the cutting edge and risk bugs in order to get new features. This tactic is not yet formally imitated by most Linux hackers, but perhaps it should be; the fact that either choice is available makes both more attractive. [HBS]

5. When Is A Rose Not A Rose?

Having studied Linus's behavior and formed a theory about why it was successful, I made a conscious decision to test this theory on my new (admittedly much less complex and ambitious) project.

But the first thing I did was reorganize and simplify popclient a lot. Carl Harris's implementation was very sound, but exhibited a kind of unnecessary complexity common to many C programmers . He treated the code as central and the data structures as support for the code. As a result, the code was beautiful but the data structure design ad-hoc and rather ugly (at least by the high standards of this old LISP hacker).

I had another purpose for rewriting besides improving the code and the data structure design, however. That was to evolve it into something I understood completely. It's no fun to be responsible for fixing bugs in a program you don't understand.

For the first month or so, then, I was simply following out the implications of Carl's basic design. The first serious change I made was to add IMAP support. I did this by reorganizing the protocol machines into a generic driver and three method tables (for POP2, POP3, and IMAP). This and the previous changes illustrate a general principle that's good for programmers to keep in mind, especially in languages like C that don't naturally do dynamic typing:

9. Smart data structures and dumb code works a lot better than the other way around.

Brooks, Chapter 9: ``Show me your [code] and conceal your [data structures], and I shall continue to be mystified. Show me your [data structures], and I won't usually need your [code]; it'll be obvious.''

Actually, he said ``flowcharts'' and ``tables''. But allowing for thirty years of terminological/cultural shift, it's almost the same point.

At this point (early September 1996, about six weeks from zero) I started thinking that a name change might be in order -- after all, it wasn't just a POP client any more. But I hesitated, because there was as yet nothing genuinely new in the design. My version of popclient had yet to develop an identity of its own.

That changed, radically, when fetchmail learned how to forward fetched mail to the SMTP port. I'll get to that in a moment. But first: I said above that I'd decided to use this project to test my theory about what Linus Torvalds had done right. How (you may well ask) did I do that? In these ways:

I released early and often (almost never less often than every ten days; during periods of intense development, once a day).

I grew my beta list by adding to it everyone who contacted me about fetchmail.

I sent chatty announcements to the beta list whenever I released, encouraging people to participate.

And I listened to my beta testers, polling them about design decisions and stroking them whenever they sent in patches and feedback.

The payoff from these simple measures was immediate. From the beginning of the project, I got bug reports of a quality most developers would kill for, often with good fixes attached. I got thoughtful criticism, I got fan mail, I got intelligent feature suggestions. Which leads to:

10. If you treat your beta-testers as if they're your most valuable resource, they will respond by becoming your most valuable resource.

One interesting measure of fetchmail's success is the sheer size of the project beta list, fetchmail-friends. At time of writing it has 249 members and is adding two or three a week.

Actually, as I revise in late May 1997 the list is beginning to lose members from its high of close to 300 for an interesting reason. Several people have asked me to unsubscribe them because fetchmail is working so well for them that they no longer need to see the list traffic! Perhaps this is part of the normal life-cycle of a mature bazaar-style project.

6. Popclient becomes Fetchmail

The real turning point in the project was when Harry Hochheiser sent me his scratch code for forwarding mail to the client machine's SMTP port. I realized almost immediately that a reliable implementation of this feature would make all the other mail delivery modes next to obsolete.

For many weeks I had been tweaking fetchmail rather incrementally while feeling like the interface design was serviceable but grubby -- inelegant and with too many exiguous options hanging out all over. The options to dump fetched mail to a mailbox file or standard output particularly bothered me, but I couldn't figure out why.

(If you don't care about the technicalia of Internet mail, the next two paragraphs can be safely skipped.)

What I saw when I thought about SMTP forwarding was that popclient had been trying to do too many things. It had been designed to be both a mail transport agent (MTA) and a local delivery agent (MDA). With SMTP forwarding, it could get out of the MDA business and be a pure MTA, handing off mail to other programs for local delivery just as sendmail does.

Why mess with all the complexity of configuring a mail delivery agent or setting up lock-and-append on a mailbox when port 25 is almost guaranteed to be there on any platform with TCP/IP support in the first place? Especially when this means retrieved mail is guaranteed to look like normal sender-initiated SMTP mail, which is really what we want anyway.

(Back to a higher level...)

Even if you didn't follow the preceding technical jargon, there are several important lessons here. First, this SMTP-forwarding concept was the biggest single payoff I got from consciously trying to emulate Linus's methods. A user gave me this terrific idea -- all I had to do was understand the implications.

11. The next best thing to having good ideas is recognizing good ideas from your users. Sometimes the latter is better.

Interestingly enough, you will quickly find that if you are completely and self-deprecatingly truthful about how much you owe other people, the world at large will treat you like you did every bit of the invention yourself and are just being becomingly modest about your innate genius. We can all see how well this worked for Linus!

(When I gave my talk at the Perl conference in August 1997, hacker extraordinaire Larry Wall was in the front row. As I got to the last line above he called out, religious-revival style, ``Tell it, tell it, brother!''. The whole audience laughed, because they knew this had worked for the inventor of Perl, too.)

After a very few weeks of running the project in the same spirit, I began to get similar praise not just from my users but from other people to whom the word leaked out. I stashed away some of that email; I'll look at it again sometime if I ever start wondering whether my life has been worthwhile :-).

But there are two more fundamental, non-political lessons here that are general to all kinds of design.

12. Often, the most striking and innovative solutions come from realizing that your concept of the problem was wrong.

I had been trying to solve the wrong problem by continuing to develop popclient as a combined MTA/MDA with all kinds of funky local delivery modes. Fetchmail's design needed to be rethought from the ground up as a pure MTA, a part of the normal SMTP-speaking Internet mail path.

When you hit a wall in development – when you find yourself hard put to think past the next patch – it's often time to ask not whether you've got the right answer, but whether you're asking the right question. Perhaps the problem needs to be reframed.

Well, I had reframed my problem. Clearly, the right thing to do was (1) hack SMTP forwarding support into the generic driver, (2) make it the default mode, and (3) eventually throw out all the other delivery modes, especially the deliver-to-file and deliver-to-standard-output options.

I hesitated over step 3 for some time, fearing to upset long-time popclient users dependent on the alternate delivery mechanisms. In theory, they could immediately switch to .forward files or their non-sendmail equivalents to get the same effects. In practice the transition might have been messy.

But when I did it, the benefits proved huge. The cruftiest parts of the driver code vanished. Configuration got radically simpler – no more grovelling around for the system MDA and user's mailbox, no more worries about whether the underlying OS supports file locking.

Also, the only way to lose mail vanished. If you specified delivery to a file and the disk got full, your mail got lost. This can't happen with SMTP forwarding because your SMTP listener won't return OK unless the message can be delivered or at least spooled for later delivery.

Also, performance improved (though not so you'd notice it in a single run). Another not insignificant benefit of this change was that the manual page got a lot simpler.

Later, I had to bring delivery via a user-specified local MDA back in order to allow handling of some obscure situations involving dynamic SLIP. But I found a much simpler way to do it.

The moral? Don't hesitate to throw away superannuated features when you can do it without loss of effectiveness. Antoine de Saint-Exupéry (who was an aviator and aircraft designer when he wasn't being the author of classic children's books) said:

13. ``Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away.''

When your code is getting both better and simpler, that is when you **know** it's right. And in the process, the fetchmail design acquired an identity of its own, different from the ancestral popclient.

It was time for the name change. The new design looked much more like a dual of sendmail than the old popclient had; both are MTAs, but where sendmail pushes then delivers, the new popclient pulls then delivers. So, two months off the blocks, I renamed it fetchmail.

There is a more general lesson in this story about how SMTP delivery came to fetchmail. It is not only debugging that is parallelizable; development and (to a perhaps surprising extent) exploration of design space is, too. When your development mode is rapidly iterative, development and enhancement may become special cases of debugging -- fixing `bugs of omission' in the original capabilities or concept of the software.

Even at a higher level of design, it can be very valuable to have the thinking of lots of co-developers random-walking through the design space near your product. Consider the way a puddle of water finds a drain, or better yet how ants find food: exploration essentially by diffusion, followed by exploitation mediated by a scalable communication mechanism. This works very well; as with Harry Hochheiser and me, one of your outriders may well find a huge win nearby that you were just a little too close-focused to see.

7. Fetchmail Grows Up

There I was with a neat and innovative design, code that I knew worked well because I used it every day, and a burgeoning beta list. It gradually dawned on me that I was no longer engaged in a trivial personal hack that might happen to be useful to few other people. I had my hands on a program every hacker with a Unix box and a SLIP/PPP mail connection really needs.

With the SMTP forwarding feature, it pulled far enough in front of the competition to potentially become a ``category killer'', one of those classic programs that fills its niche so competently that the alternatives are not just discarded but almost forgotten.

I think you can't really aim or plan for a result like this. You have to get pulled into it by design ideas so powerful that afterward the results just seem inevitable, natural, even foreordained. The only way to try for ideas like that is by having lots of ideas -- or by having the engineering judgment to take other peoples' good ideas beyond where the originators thought they could go.

Andy Tanenbaum had the original idea to build a simple native Unix for IBM PCs, for use as a teaching tool (he called it Minix). Linus Torvalds pushed the Minix concept further than Andrew probably thought it could go -- and it grew into something wonderful. In the same way (though on a smaller scale), I took some ideas by Carl Harris and Harry Hochheiser and pushed them hard. Neither of us was `original' in the romantic way people think is genius. But then, most science and engineering and software development isn't done by original genius, hacker mythology to the contrary.

The results were pretty heady stuff all the same -- in fact, just the kind of success every hacker lives for! And they meant I would have to set my standards even higher. To make fetchmail as good as I now saw it could be, I'd have to write not just for my own needs, but also include and support features necessary to others but outside my orbit. And do that while keeping the program simple and robust.

The first and overwhelmingly most important feature I wrote after realizing this was multidrop support -- the ability to fetch mail from mailboxes that had accumulated all mail for a group of users, and then route each piece of mail to its individual recipients.

I decided to add the multidrop support partly because some users were clamoring for it, but mostly because I thought it would shake bugs out of the single-drop code by forcing me to deal with addressing in full generality. And so it proved. Getting RFC 822 address parsing right took me a remarkably long time, not because any individual piece of it is hard but because it involved a pile of interdependent and fussy details.

But multidrop addressing turned out to be an excellent design decision as well. Here's how I knew:

14. Any tool should be useful in the expected way, but a truly great tool lends itself to uses you never expected.

The unexpected use for multi-drop fetchmail is to run mailing lists with the list kept, and alias expansion done, on the **client** side of the Internet connection. This means someone running a personal machine through an ISP account can manage a mailing list without continuing access to the ISP's alias files.

Another important change demanded by my beta testers was support for 8-bit MIME (Multipurpose Internet Mail Extensions) operation. This was pretty easy to do, because I had been careful to keep the code 8-bit clean. Not because I anticipated the demand for this feature, but rather in obedience to another rule:

15. When writing gateway software of any kind, take pains to disturb the data stream as little as possible -- and *never* throw away information unless the recipient forces you to!

Had I not obeyed this rule, 8-bit MIME support would have been difficult and buggy. As it was, all I had to do is read the MIME syandard (RFC 1652) and add a trivial bit of header-generation logic.

Some European users bugged me into adding an option to limit the number of messages retrieved per session (so they can control costs from their expensive phone networks). I resisted this for a long time, and I'm still not entirely happy about it. But if you're writing for the world, you have to listen to your customers -- this doesn't change just because they're not paying you in money.

8. A Few More Lessons From Fetchmail

Before we go back to general software-engineering issues, there are a couple more specific lessons from the fetchmail experience to ponder. Nontechnical readers can safely skip this section.

The rc (control) file syntax includes optional `noise' keywords that are entirely ignored by the parser. The English-like syntax they allow is considerably more readable than the traditional terse keyword-value pairs you get when you strip them all out.

These started out as a late-night experiment when I noticed how much the rc file declarations were beginning to resemble an imperative minilanguage. (This is also why I changed the original popclient `server' keyword to `poll').

It seemed to me that trying to make that imperative minilanguage more like English might make it easier to use. Now, although I'm a convinced partisan of the ``make it a language'' school of design as exemplified by Emacs and HTML and many database engines, I am not normally a big fan of ``English-like'' syntaxes.

Traditionally programmers have tended to favor control syntaxes that are very precise and compact and have no redundancy at all. This is a cultural legacy from when computing resources were expensive, so parsing stages had to be as cheap and simple as possible. English, with about 50% redundancy, looked like a very inappropriate model then.

This is not my reason for normally avoiding English-like syntaxes; I mention it here only to demolish it. With cheap cycles and core, terseness should not be an end in itself. Nowadays it's more important for a language to be convenient for humans than to be cheap for the computer.

There remain, however, good reasons to be wary. One is the complexity cost of the parsing stage -- you don't want to raise that to the point where it's a significant source of bugs and user confusion in itself. Another is that trying to make a language syntax English-like often demands that the ``English'' it speaks be bent seriously out of shape, so much so that the superficial resemblance to natural language is as confusing as a traditional syntax would have been. (You see this bad effect in a lot of so-called ``fourth generation'' and commercial database-query languages.)

The fetchmail control syntax seems to avoid these problems because the language domain is extremely restricted. It's nowhere near a general-purpose language; the things it says simply are not very complicated, so there's little potential for confusion in moving mentally between a tiny subset of English and the actual control language. I think there may be a wider lesson here:

16. When your language is nowhere near Turing-complete, syntactic sugar can be your friend.

Another lesson is about security by obscurity. Some fetchmail users asked me to change the software to store passwords encrypted in the rc file, so snoopers wouldn't be able to casually see them.

I didn't do it, because this doesn't actually add protection. Anyone who's acquired permissions to read your rc file will be able to run fetchmail as you anyway -- and if it's your password

they're after, they'd be able to rip the necessary decoder out of the fetchmail code itself to get it.

All .fetchmailrc password encryption would have done is give a false sense of security to people who don't think very hard. The general rule here is:

17. A security system is only as secure as its secret. Beware of pseudo-secrets.

9. Necessary Preconditions for the Bazaar Style

Early reviewers and test audiences for this paper consistently raised questions about the preconditions for successful bazaar-style development, including both the qualifications of the project leader and the state of code at the time one goes public and starts to try to build a co-developer community.

It's fairly clear that one cannot code from the ground up in bazaar style [IN] . One can test, debug and improve in bazaar style, but it would be very hard to **originate** a project in bazaar mode. Linus didn't try it. I didn't either. Your nascent developer community needs to have something runnable and testable to play with.

When you start community-building, what you need to be able to present is a **plausible promise**. Your program doesn't have to work particularly well. It can be crude, buggy, incomplete, and poorly documented. What it must not fail to do is (a) run, and (b) convince potential co-developers that it can be evolved into something really neat in the foreseeable future.

Linux and fetchmail both went public with strong, attractive basic designs. Many people thinking about the bazaar model as I have presented it have correctly considered this critical, then jumped from it to the conclusion that a high degree of design intuition and cleverness in the project leader is indispensable.

But Linus got his design from Unix. I got mine initially from the ancestral popclient (though it would later change a great deal, much more proportionately speaking than has Linux). So does the leader/coordinator for a bazaar-style effort really have to have exceptional design talent, or can he get by on leveraging the design talent of others?

I think it is not critical that the coordinator be able to originate designs of exceptional brilliance, but it is absolutely critical that the coordinator be able to **recognize good design ideas from others**.

Both the Linux and fetchmail projects show evidence of this. Linus, while not (as previously discussed) a spectacularly original designer, has displayed a powerful knack for recognizing good design and integrating it into the Linux kernel. And I have already described how the single most powerful design idea in fetchmail (SMTP forwarding) came from somebody else.

Early audiences of this paper complimented me by suggesting that I am prone to undervalue design originality in bazaar projects because I have a lot of it myself, and therefore take it for granted. There may be some truth to this; design (as opposed to coding or debugging) is certainly my strongest skill.

But the problem with being clever and original in software design is that it gets to be a habit -- you start reflexively making things cute and complicated when you should be keeping them robust and simple. I have had projects crash on me because I made this mistake, but I managed not to with fetchmail.

So I believe the fetchmail project succeeded partly because I restrained my tendency to be clever; this argues (at least) against design originality being essential for successful bazaar projects. And consider Linux. Suppose Linus Torvalds had been trying to pull off fundamental innovations in operating system design during the development; does it seem at all likely that the resulting kernel would be as stable and successful as what we have?

A certain base level of design and coding skill is required, of course, but I expect almost anybody seriously thinking of launching a bazaar effort will already be above that minimum. The open-source community's internal market in reputation exerts subtle pressure on people not to launch development efforts they're not competent to follow through on. So far this seems to have worked pretty well.

There is another kind of skill not normally associated with software development which I think is as important as design cleverness to bazaar projects -- and it may be more important. A bazaar project coordinator or leader must have good people and communications skills.

This should be obvious. In order to build a development community, you need to attract people, interest them in what you're doing, and keep them happy about the amount of work they're doing. Technical sizzle will go a long way towards accomplishing this, but it's far from the whole story. The personality you project matters, too.

It is not a coincidence that Linus is a nice guy who makes people like him and want to help him. It's not a coincidence that I'm an energetic extrovert who enjoys working a crowd and has some of the delivery and instincts of a stand-up comic. To make the bazaar model work, it helps enormously if you have at least a little skill at charming people.

10. The Social Context of Open-Source Software

It is truly written: the best hacks start out as personal solutions to the author's everyday problems, and spread because the problem turns out to be typical for a large class of users. This takes us back to the matter of rule 1, restated in a perhaps more useful way:

18. To solve an interesting problem, start by finding a problem that is interesting to you.

So it was with Carl Harris and the ancestral popclient, and so with me and fetchmail. But this has been understood for a long time. The interesting point, the point that the histories of Linux and fetchmail seem to demand we focus on, is the next stage -- the evolution of software in the presence of a large and active community of users and co-developers.

In ``The Mythical Man-Month'', Fred Brooks observed that programmer time is not fungible; adding developers to a late software project makes it later. He argued that the complexity and communication costs of a project rise with the square of the number of developers, while work done only rises linearly. This claim has since become known as ``Brooks's Law'' and is widely regarded as a truism. But if Brooks's Law were the whole picture, Linux would be impossible.

Gerald Weinberg's classic ``The Psychology Of Computer Programming'' supplied what, in hindsight, we can see as a vital correction to Brooks. In his discussion of ``egoless programming'', Weinberg observed that in shops where developers are not territorial about their code, and encourage

other people to look for bugs and potential improvements in it, improvement happens dramatically faster than elsewhere.

Weinberg's choice of terminology has perhaps prevented his analysis from gaining the acceptance it deserved -- one has to smile at the thought of describing Internet hackers as ``egoless''. But I think his argument looks more compelling today than ever.

The history of Unix should have prepared us for what we're learning from Linux (and what I've verified experimentally on a smaller scale by deliberately copying Linus's methods [EGCS]). That is, that while coding remains an essentially solitary activity, the really great hacks come from hamessing the attention and brainpower of entire communities. The developer who uses only his or her own brain in a closed project is going to fall behind the developer who knows how to create an open, evolutionary context in which feedback exploring the design space, code contributions, bug-spotting, and other improvements come back from hundreds (perhaps thousands) of people.

But the traditional Unix world was prevented from pushing this approach to the ultimate by several factors. One was the legal contraints of various licenses, trade secrets, and commercial interests. Another (in hindsight) was that the Internet wasn't yet good enough.

Before cheap Internet, there were some geographically compact communities where the culture encouraged Weinberg's ``egoless'' programming, and a developer could easily attract a lot of skilled kibitzers and co-developers. Bell Labs, the MIT AI Lab, UC Berkeley – these became the home of innovations that are legendary and still potent.

Linux was the first project to make a conscious and successful effort to use the entire **world** as its talent pool. I don't think it's a coincidence that the gestation period of Linux coincided with the birth of the World Wide Web, and that Linux left its infancy during the same period in 1993-1994 that saw the takeoff of the ISP industry and the explosion of mainstream interest in the Internet. Linus was the first person who learned how to play by the new rules that pervasive Internet access made possible.

While cheap Internet was a necessary condition for the Linux model to evolve, I think it was not by itself a sufficient condition. Another vital factor was the development of a leadership style and set of cooperative customs that could allow developers to attract co-developers and get maximum leverage out of the medium.

But what is this leadership style and what are these customs? They cannot be based on power relationships -- and even if they could be, leadership by coercion would not produce the results we see. Weinberg quotes the autobiography of the 19th-century Russian anarchist Pyotr Alexeyvich Kropotkin's ``Memoirs of a Revolutionist'' to good effect on this subject:

``Having been brought up in a serf-owner's family, I entered active life, like all young men of my time, with a great deal of confidence in the necessity of commanding, ordering, scolding, punishing and the like. But when, at an early stage, I had to manage serious enterprises and to deal with [free] men, and when each mistake would lead at once to heavy consequences, I began to appreciate the difference between acting on the principle of command and discipline and acting on the principle of common understanding. The former works admirably in a military parade, but it is worth nothing where real life is concerned, and the aim can be achieved only through the severe effort of many converging wills.''

The ``severe effort of many converging wills'' is precisely what a project like Linux requires -- and the ``principle of command'' is effectively impossible to apply among volunteers in the anarchist's paradise we call the Internet. To operate and compete effectively, hackers who want to lead collaborative projects have to learn how to recruit and energize effective communities of interest in the mode vaguely suggested by Kropotkin's ``principle of understanding''. They must learn to use Linus's Law. [SP]

Earlier I referred to the ``Delphi effect'' as a possible explanation for Linus's Law. But more powerful analogies to adaptive systems in biology and economics also irresistably suggest themselves. The Linux world behaves in many respects like a free market or an ecology, a collection of selfish agents attempting to maximize utility which in the process produces a self-correcting spontaneous order more elaborate and efficient than any amount of central planning could have achieved. Here, then, is the place to seek the ``principle of understanding''.

The ``utility function'' Linux hackers are maximizing is not classically economic, but is the intangible of their own ego satisfaction and reputation among other hackers. (One may call their motivation ``altruistic'', but this ignores the fact that altruism is itself a form of ego satisfaction for the altruist). Voluntary cultures that work this way are not actually uncommon; one other in which I have long participated is science fiction fandom, which unlike hackerdom has long explicitly recognized ``egoboo'' (ego-boosting, or the enhancement of one's reputation among other fans) as the basic drive behind volunteer activity.

Linus, by successfully positioning himself as the gatekeeper of a project in which the development is mostly done by others, and nurturing interest in the project until it became self-sustaining, has shown an acute grasp of Kropotkin's ``principle of shared understanding''. This quasi-economic view of the Linux world enables us to see how that understanding is applied.

We may view Linus's method as a way to create an efficient market in ``egoboo'' -- to connect the selfishness of individual hackers as firmly as possible to difficult ends that can only be achieved by sustained cooperation. With the fetchmail project I have shown (albeit on a smaller scale) that his methods can be duplicated with good results. Perhaps I have even done it a bit more consciously and systematically than he.

Many people (especially those who politically distrust free markets) would expect a culture of self-directed egoists to be fragmented, territorial, wasteful, secretive, and hostile. But this expectation is clearly falsified by (to give just one example) the stunning variety, quality and depth of Linux documentation. It is a hallowed given that programmers **hate** documenting; how is it, then, that Linux hackers generate so much of it? Evidently Linux's free market in egoboo works better to produce virtuous, other-directed behavior than the massively-funded documentation shops of commercial software producers.

Both the fetchmail and Linux kernel projects show that by properly rewarding the egos of many other hackers, a strong developer/coordinator can use the Internet to capture the benefits of having lots of co-developers without having a project collapse into a chaotic mess. So to Brooks's Law I counter-propose the following:

19: Provided the development coordinator has a medium at least as good as the Internet, and knows how to lead without coercion, many heads are inevitably better than one.

I think the future of open-source software will increasingly belong to people who know how to play Linus's game, people who leave

behind the cathedral and embrace the bazaar. This is not to say that individual vision and brilliance will no longer matter; rather, I think that the cutting edge of open-source software will belong to people who start from individual vision and brilliance, then amplify it through the effective construction of voluntary communities of interest.

Perhaps this is not only the future of **open-source** software. No closed-source developer can match the pool of talent the Linux community can bring to bear on a problem. Very few could afford even to hire the more than two hundred (1999: six hundred, 2000: eight hundred) people who have contributed to fetchmail!

Perhaps in the end the open-source culture will triumph not because cooperation is morally right or software ``hoarding'' is morally wrong (assuming you believe the latter, which neither Linus nor I do), but simply because the closed-source world cannot win an evolutionary arms race with open-source communities that can put orders of magnitude more skilled time into a problem.

11. On Management and the Maginot Line

The original ``Cathedral and Bazaar'' paper ended with the vision above -- that of happy networked hordes of programmer/anarchists outcompeting and overwhelming the hierarchical world of conventional closed software.

A good many skeptics weren't convinced, however; and the questions they raise deserve a fair engagement. Most of the objections to the bazaar argument come down to the claim that its proponents have underestimated the productivity-multiplying effect of conventional management.

Traditionally-minded software-development managers often object that the casualness with which project groups form and change and dissolve in the open-source world negates a significant part of the apparent advantage of numbers that the open-source community has over any single closed-source developer. They would observe that in software development it is really sustained effort over time and the degree to which customers can expect continuing investment in the product that matters, not just how many people have thrown a bone in the pot and left it to simmer.

There is something to this argument, to be sure; in fact, I have developed the idea that expected future service value is the key to the economics of software production in The Magic Cauldron.

But this argument also has a major hidden problem; its implicit assumption that open-source development cannot deliver such sustained effort. In fact, there have been open-source projects that maintained a coherent direction and an effective maintainer community over quite long periods of time without the kinds of incentive structures or institutional controls that conventional management finds essential. The development of the GNU Emacs editor is an extreme and instructive example; it has absorbed the efforts of hundreds of contributors over fifteen years into a unified architectural vision, despite high turnover and the fact that only one person (its author) has been continuously active during all that time. No closed-source editor has ever matched this longevity record.

This suggests a reason for questioning the advantages of conventionally-managed software development that is independent of the rest of the arguments over cathedral vs. bazaar mode. If it's possible for GNU Emacs to express a consistent architectural vision over fifteen years, or for an operating system like Linux to do the same over eight years of rapidly changing hardware and platform technology; and if (as is indeed the case) there have been many well-architected open-source projects of more than five years duration -- then we are entitled to wonder what, if anything, the tremendous overhead of conventionally-managed development is actually buying us.

Whatever it is certainly doesn't include reliable execution by deadline, or on budget, or to all features of the specification; it's a rare `managed' project that meets even one of these goals, let alone all three. It also does not appear to be ability to adapt to changes in technology and economic context during the project lifetime, either; the open-source community has proven **far** more effective on that score (as one can readily verify, for example, by comparing the thirty-year history of the Internet with the short half-lives of proprietary networking technologies -- or the cost of the 16-bit to 32-bit transition in Microsoft Windows with the nearly effortless up-migration of Linux during the same period, not only along the Intel line of development but to more than a dozen other hardware platforms including the 64-bit Alpha as well).

One thing many people think the traditional mode buys you is somebody to hold legally liable and potentially recover compensation from if the project goes wrong. But this is an illusion; most software licenses are written to disclaim even warranty of merchantability, let alone performance -- and cases of successful recovery for software nonperformance are vanishingly rare. Even if they were common, feeling comforted by having somebody to sue would be missing the point. You didn't want to be in a lawsuit; you wanted working software.

So what is all that management overhead buying?

In order to understand that, we need to understand what software development managers believe they do. A woman I know who seems to be very good at this job says software project management has five functions:

To **define goals** and keep everybody pointed in the same direction.

To **monitor** and make sure crucial details don't get skipped.

To **motivate** people to do boring but necessary drudgework.

To **organize** the deployment of people for best productivity.

To **marshal resources** needed to sustain the project.

Apparently worthy goals, all of these; but under the open-source model, and in its surrounding social context, they can begin to seem strangely irrelevant. We'll take them in reverse order.

My friend reports that a lot of **resource marshalling** is basically defensive; once you have your people and machines and office space, you have to defend them from peer managers competing for the same resources, and higher-ups trying to allocate the most efficient use of a limited pool.

But open-source developers are volunteers, self-selected for both interest and ability to contribute to the projects they work on (and this remains generally true even when they are being paid a salary to hack open source.) The volunteer ethos tends to take care of the `attack' side of resource-marshalling automatically; people bring their own resources to the table. And there is little or no need for a manager to `play defense' in the conventional sense.

Anyway, in a world of cheap PCs and fast Internet links, we find pretty consistently that the only really limiting resource is skilled attention. Open-source projects, when they founder, essentially never do so for want of machines or links or office space; they die only when the developers themselves lose interest.

That being the case, it's doubly important that open-source hackers **organize themselves** for maximum productivity by self-selection -- and the social milieu selects ruthlessly for competence. My friend, familiar with both the open-source world and large closed projects, believes that open source has been successful partly because its culture only accepts the most talented 5% or so of the programming population. She spends most of her time organizing the deployment of the other 95%, and has thus observed first-hand the well-known variance of a factor of one hundred in productivity between the most able programmers and the merely competent.

The size of that variance has always raised an awkward question: would individual projects, and the field as a whole, be better off without more than 50% of the least able in it? Thoughtful managers have understood for a long time that if conventional software management's only function were to convert the least able from a net loss to a marginal win, the game might not be worth the candle.

The success of the open-source community sharpens this question considerably, by providing hard evidence that it is often cheaper and more effective to recruit self-selected volunteers from the Internet than it is to manage buildings full of people who would rather be doing something else.

Which brings us neatly to the question of **motivation**. An equivalent and often-heard way to state my friend's point is that traditional development management is a necessary compensation for poorly motivated programmers who would not otherwise turn out good work.

This answer usually travels with a claim that the open-source community can only be relied on to do work that is `sexy' or technically sweet; anything else will be left undone (or done only poorly) unless it's churned out by money-motivated cubicle peons with managers cracking whips over them. I address the psychological and social reasons for being skeptical of this claim in ``Homesteading the muttNoosphere''. For present purposes, however, I think it's more interesting to point out the implications of accepting it as true.

If the conventional, closed-source, heavily-managed style of software development is really defended only by a sort of Maginot line of problems conducive to boredom, then it's going to remain viable in each individual application area for only so long as nobody finds those problems really interesting and nobody else finds any way to route around them. Because the moment there is open-source competition for a `boring' piece of software, customers are going to know that it was finally tackled by someone who chose that problem to solve because of a fascination with the problem itself -- which, in software as in other kinds of creative work, is a far more effective motivator than money alone.

Having a conventional management structure solely in order to motivate, then, is probably good tactics but bad strategy; a short-term win, but in the longer term a surer loss.

So far, conventional development management looks like a bad bet now against open source on two points (resource marshalling, organization), and like it's living on borrowed time with respect to a third (motivation). And the poor beleaguered conventional manager is not going to get any succour from the **monitoring** issue; the strongest argument the open-source community has is that decentralized peer review trumps all the conventional methods for trying to ensure that details don't get slipped.

Can we save **defining goals** as a justification for the overhead of conventional software project management? Perhaps; but to do so, we'll need good reason to believe that management committees and corporate roadmaps are more successful at defining worthy and widely-shared goals than the project leaders and tribal elders who fill the analogous role in the open-source world.

That is on the face of it a pretty hard case to make. And it's not so much the open-source side of the balance (the longevity of Emacs, or Linus Torvalds's ability to mobilize hordes of developers with talk of ``world domination'') that makes it tough. Rather, it's the demonstrated awfulness of conventional mechanisms for defining the goals of software projects.

One of the best-known folk theorems of software engineering is that 60% to 75% of conventional software projects either are never completed or are rejected by their intended users. If that range is anywhere near true (and I've never met a manager of any experience who disputes it) then more projects than not are being aimed at goals that are either (a) not realistically attainable, or (b) just plain wrong.

This, more than any other problem, is the reason that in today's software engineering world the very phrase ``management committee'' is likely to send chills down the hearer's spine – even (or perhaps especially) if the hearer is a manager. The days when only programmers griped about this pattern are long past; `Dilbert' cartoons hang over **executives'** desks now.

Our reply, then, to the traditional software development manager, is simple – if the open-source community has really underestimated the value of conventional management, **why do so many of you display contempt for your own process?**

Once again the existence of the open-source community sharpens this question considerably -- because we have **fun** doing what we do. Our creative play has been racking up technical, market-share, and mind-share successes at an astounding rate. We're proving not only that we can do better software, but that **joy is an asset**.

Two and a half years after the first version of this essay, the most radical thought I can offer to close with is no longer a vision of an open-source-dominated software world; that, after all, looks plausible to a lot of sober people in suits these days.

Rather, I want to suggest what may be a wider lesson about software, (and probably about every kind of creative or professional work). Human beings generally take pleasure in a task when it falls in a sort of optimal-challenge zone; not so easy as to be boring, not too hard to achieve. A happy programmer is one who is neither underutilized nor weighed down with ill-formulated goals and stressful process friction. **Enjoyment predicts efficiency.**

Relating to your own work process with fear and loathing (even in the displaced, ironic way suggested by hanging up Dilbert cartoons) should therefore be regarded in itself as a sign that the process has failed. Joy, humor, and playfulness are indeed assets; it was not mainly for the alliteration that I wrote of "happy hordes" above, and it is no mere joke that the Linux mascot is a cuddly, neotenous penguin.

It may well turn out that one of the most important effects of open source's success will be to teach us that play is the most economically efficient mode of creative work.

12. Acknowledgements

This paper was improved by conversations with a large number of people who helped debug it. Particular thanks to Jeff Dutky <dutky@wam.umd.edu>, who suggested the ``debugging is parallelizable'' formulation, and helped develop the analysis that proceeds from it. Also to Nancy Lebovitz <nancyl@universe.digex.net> for her suggestion that I emulate Weinberg by quoting Kropotkin. Perceptive criticisms also came from Joan Eslinger <wombat@kilimanjaro.engr.sgi.com> and Marty Franz <marty@net-link.net> of the General Technics list. Glen Vandenburg <glv@vanderburg.org> pointeed out the importance of self-selection in contributor populations and suggested the fruitful idea that much development rectifies `bugs of omission'; Daniel Upper <upper@peak.org> suggested the natural analogies for this. I'm grateful to the members of PLUG, the Philadelphia Linux User's group, for providing the first test audience for the first public version of this paper. Paula Matuszek <matusp00@mh.us.sbphrd.com> enlightened me about the practice of software management. Phil Hudson <phil.hudson@iname.com> reminded me that the social organization of the hacker culture mirrors the organization of its software, and vice-versa. Finally, Linus Torvalds's comments were helpful and his early endorsement very encouraging.

13. For Further Reading

I quoted several bits from Frederick P. Brooks's classic **The Mythical Man-Month** because, in many respects, his insights have yet to be improved upon. I heartily recommend the 25th Anniversary edition from Addison-Wesley (ISBN 0-201-83595-9), which adds his 1986 **``No Silver Bullet''** paper.

The new edition is wrapped up by an invaluable 20-years-later retrospective in which Brooks forthrightly admits to the few judgements in the original text which have not stood the test of time. I first read the retrospective after the first public version of this paper was substantially complete, and was surprised to discover that Brooks attributes bazaar-like practices to Microsoft! (In fact, however, this attribution turned out to be mistaken. In 1998 we learned from the Halloween Documents that Microsoft's internal developer community is heavily balkanized, with the kind of general source access needed to support a bazaar not even truly possible.)

Gerald M. Weinberg's **The Psychology Of Computer Programming** (New York, Van Nostrand Reinhold 1971) introduced the rather unfortunately-labeled concept of ``egoless programming''. While he was nowhere near the first person to realize the futility of the ``principle of command'', he was probably the first to recognize and argue the point in particular connection with software development.

Richard P. Gabriel, contemplating the Unix culture of the pre-Linux era, reluctantly argued for the superiority of a primitive bazaar-like model in his 1989 paper **Lisp: Good News, Bad News, and How To Win Big**. Though dated in some respects, this essay is still rightly celebrated among Lisp fans (including me). A correspondent reminded me that the section titled ``Worse Is Better'' reads almost as an anticipation of Linux. The paper is accessible on the World Wide Web at http://www.naggum.no/worse-is-better.html.

De Marco and Lister's **Peopleware: Productive Projects and Teams** (New York; Dorset House, 1987; ISBN 0-932633-05-6) is an underappreciated gem which I was delighted to see Fred Brooks cite in his retrospective. While little of what the authors have to say is directly applicable to the Linux or open-source communities, the authors' insight into the conditions necessary for creative work is acute and worthwhile for anyone attempting to import some of the bazaar model's virtues into a commercial context.

Finally, I must admit that I very nearly called this paper ``The Cathedral and the Agora'', the latter term being the Greek for an open market or public meeting place. The seminal ``agoric systems'' papers by Mark Miller and Eric Drexler, by describing the emergent properties of market-like computational ecologies, helped prepare me to think clearly about analogous phenomena in the open-source culture when Linux rubbed my nose in them five years later. These papers are available on the Web at http://www.agorics.com/agorpapers.html.

14. Epilog: Netscape Embraces the Bazaar

It's a strange feeling to realize you're helping make history....

On January 22 1998, approximately seven months after I first published ``The Cathedral and the Bazaar'', Netscape Communications, Inc. announced plans to give away the source for Netscape Communicator. I had had no clue this was going to happen before the day of the announcement.

Eric Hahn, Executive Vice President and Chief Technology Officer at Netscape, emailed me shortly afterwards as follows: ``On behalf of everyone at Netscape, I want to thank you for helping us get to this point in the first place. Your thinking and writings were fundamental inspirations to our decision.''

The following week I flew out to Silicon Valley at Netscape's invitation for a day-long strategy conference (on Feb 4 1998) with some of their top executives and technical people. We designed Netscape's source-release strategy and license together.

A few days later I wrote the following:

"Netscape is about to provide us with a large-scale, real-world test of the bazaar model in the commercial world. The open-source culture now faces a danger; if Netscape's execution doesn't work, the open-source concept may be so discredited that the commercial world won't touch it again for another decade.

On the other hand, this is also a spectacular opportunity. Initial reaction to the move on Wall Street and elsewhere has been cautiously positive. We're being given a chance to prove ourselves, too. If Netscape regains substantial market share through this move, it just may set off a long-overdue revolution in the software industry.

The next year should be a very instructive and interesting time."

And indeed it was. As I write in mid-1999, the development of what was later named `Mozilla' has been only a qualified success. It achieved Netscape's original goal, which was to deny Microsoft a monopoly lock on the browser market. It has also achieved some dramatic successes (notably the release of the next-generation Gecko rendering engine).

However, it has not yet garnered the massive development effort from outside Netscape that the Mozilla founders had originally hoped for. The problem here seems to be that for a long time the Mozilla distribution actually broke one of the basic rules of the bazaar model; they didn't ship something potential contributors could easily run and see working. (Until more than a year after release, building Mozilla from source required a license for the proprietary Motif library.)

Most negatively (from the point of view of the outside world) the Mozilla group has yet to ship a production-quality browser -- and

one of the project's principals caused a bit of a sensation by resigning, complaining of poor management and missed opportunities. ``Open source,'' he correctly observed, ``is not magic pixie dust.''

And indeed it is not. The long-term prognosis for Mozilla looks rather better now (in August 1999) than it did at the time of Jamie Zawinski's resignation letter -- but he was right to point out that going open will not necessarily save an existing project that suffers from ill-defined goals or spaghetti code or any of the software engineering's other chronic ills. Mozilla has managed to provide an example simultaneously of how open source can succeed and how it could fail.

In the mean time, however, the open-source idea has scored successes and found backers elsewhere. 1998 and late 1999 saw a tremendous explosion of interest in the open-source development model, a trend both driven by and driving the continuing success of the Linux operating system. The trend Mozilla touched off is continuing at an accelerating rate.

15. Endnotes

[JB] In **Programing Pearls**, the noted computer-science aphorist Jon Bentley comments on Brooks's observation with ``If you plan to throw one away, you will throw away two.''. He is almost certainly right. The point of Brooks's observation, and Bentley's, isn't merely that you should expect first attempt to be wrong, it's that starting over with the right idea is usually more effective than trying to salvage a mess.

[QR] Examples of successful open-source, bazaar development predating the Internet explosion and unrelated to the Unix and Internet traditions have existed. The development of the info-Zip compression utility during 1990-1992, primarily for DOS machines, was one such. Another was the RBBS bulletin board system (again for DOS), which began in 1983 and developed a sufficiently strong community that there have been fairly regular releases up to the present (mid-1999) despite the huge technical advantages of Internet mail and file-sharing over local BBSs. While the info-Zip community relied to some extent on Internet mail, the RBBS developer culture was actually able to base a substantial on-line community on RBBS that was completely independent of the TCP/IP infrastructure.

[JH] John Hasler has suggested an interesting explanation for the fact that duplication of effort doesn't seem to be a net drag on open-source development. He proposes what I'll dub ``Hasler's Law'': the costs of duplicated work tend to scale sub-qadratically with team size -- that is, more slowly than the planning and management overhead that would be needed to eliminate them.

This claim actually does not contradict Brooks's Law. It may be the case that total complexity overhead and vulnerability to bugs scales with the square of team size, but that the costs from **duplicated** work are nevertheless a special case that scales more slowly. It's not hard to develop plausible reasons for this, starting with the undoubted fact that it is much easier to agree on functional boundaries between different developers' code that will prevent duplication of effort than it is to prevent the kinds of unplanned bad interactions across the whole system that underly most bugs.

The combination of Linus's Law and Hasler's Law suggests that there are actually three critical size regimes in software projects. On small projects (I would say one to at most three developers) no management structure more elaborate than picking a lead programmer is needed. And there is some intermediate range above that in which the cost of traditional management is relatively low, so its benefits from avoiding duplication of effort, bug-tracking, and pushing to see that details are not overlooked actually net out positive.

Above that, however, the combination of Linus's Law and Hasler's Law suggests there is a large-project range in which the costs and problems of traditional management rise much faster than the expected cost from duplication of effort. Not the least of these costs is a structural inability to harness the many-eyeballs effect, which (as we've seen) seems to do a much better job than traditional management at making sure bugs and details are not overlooked. Thus, in the large-project case, the combination of these laws effectively drives the net payoff of traditional management to zero.

[HBS] The split between Linux's experimental and stable versions has another function related to, but distinct from, hedging risk. The split attacks anither problem: the deadliness of deadlines. When programmers are held both to an immutable feature list and a fixed drop-dead date, quality goes out the window and there is likely a colossal mess in the making. I am indebted to Marco Iansiti and Alan MacCormack of the Harvard Business School for pointing me at evidence that relaxing either one of these constraints can make scheduling workable.

One way to do this is to fix the deadline but leave the feature list flexible, allowing features to drop off if not completed by deadline. This is essentially the strategy of the "stable" kernel branch; Alan Cox (the stable-kernel maintainer) puts out releases at fairly regular intervals, but makes no guarantees about when particular bugs will be fixed or features back-ported from the experimental branch.

The other way to do this is to set a desired feature list and deliver only when it is done. This is essentially the strategy of the "experimental" kernel branch. De Marco and Lister cited research showing that this scheduling policy ("wake me up when it's done") produces not only the highest quality but, on average, shorter delivery times than either "realistic" or "aggressive" scheduling.

I have come to suspect (as of early 2000) that in earlier versions of this paper I severely underestimated the importance of the "wake me up when it's done" anti-dealine policy to the open-source community's productivity and quality. General experience with the rushed GNOME 1.0 in 1999 suggests that pressure for a premature release can neutralize many of the quality benefits open source normally confers.

It may well turn out to be that the process transparency of open source is one of three coequal drivers of its quality, along with "wake me up when it's done" scheduling and developer self-selection.

[IN] An issue related to whether one can start projects from zero in the bazaar style is whether the bazaar style is capable of supporting truly innovative work. Some claim that, lacking strong leadership, the bazaar can only handle the cloning and improvement of ideas already present at the engineering state of the art, but is unable to push the state of the art. This argument was perhaps most infamously made by the Halloween Documents, two embarrassing internal Microsoft memoranda written about the open-source phenomenon. The authors compared Linux's development of a Unix-like operating system to ``chasing taillights'', and opined ``(once a project has achieved "parity" with the state-of-the-art), the level of management necessary to push towards new frontiers becomes massive.''

There are serious errors of fact implied in this argument. One is exposed when the Halloween authors themses elves later

observe that ``often [...] new research ideas are first implemented and available on Linux before they are available / incorporated into other platforms.''

If we read ``open source'' for ``Linux'', we see that this is far from a new phenomenon. Historically, the open-source community did not invent Emacs or the World Wide Web or the Internet itself by chasing taillights or being massively managed – and in the present, there is so much innovative work going on in open source that one is spoiled for choice. The GNOME project (to pick one of many) is pushing the state of the art in GUIs and object technology hard enough to have attracted considerable notice in the computer trade press well outside the Linux community. Other examples are legion, as a visit to Freshmeat on any given day will quickly prove.

But there is a more fundamental error in the implicit assumption that the **cathedral model** (or the bazaar model, or any other kind of management structure) can somehow make innovation happen reliably. This is nonsense. Gangs don't have breakthrough insights -- even volunteer groups of bazaar anarchists are usually incapable of genuine originality, let alone corporate committees of people with a survival stake in some status quo ante. **Insight comes from individuals.** The most their surrounding social machinery can ever hope to do is to be **responsive** to breakthrough insights -- to nourish and reward and rigorously test them instead of squashing them.

Some will characterize this as a romantic view, a reversion to outmoded lone-inventor stereotypes. Not so; I am not asserting that groups are incapable of **developing** breakthrough insights once they have been hatched; indeed, we learn from the peer-review process that such development groups are essential to producing a high-quality result. Rather I am pointing out that every such group development starts from -- is necessarily sparked by -- one good idea in one person's head. Cathedrals and bazaars and other social structures can catch that lightning and refine it, but they cannot make it on demand.

Therefore the root problem of innovation (in software, or anywhere else) is indeed how not to squash it -- but, even more fundamentally, it is **how to grow lots of people who can have insights in the first place**.

To suppose that cathedral-style development could manage this trick but the low entry barriers and process fluidity of the bazaar cannot would be absurd. If what it takes is one person with one good idea, then a social milieu in which one person can rapidly attract the cooperation of hundreds or thousands of others with that good idea is going inevitably to out-innovate any in which the person has to do a political sales job to a hierarchy before he can work on his idea without risk of getting fired.

And, indeed, if we look at the history of software innovation by organizations using the cathedral model, we quickly find it is rather rare. Large corporations rely on university research for new ideas (thus the Halloween Documents authors' unease about Linux's facility at coopting that research more rapidly). Or they buy out small companies built around some innovator's brain. In neither case is the innovation native to the cathedral culture; indeed, many innovations so imported end up being quietly suffocated under the "massive level of management" the Halloween Documents' authors so extol.

That, however, is a negative point. The reader would be better served by a positive one. I suggest, as an experiment, the following;

Pick a criterion for originality that you believe you can apply consistently. If your definition is ``I know it when I see it'', that's not a problem for purposes of this test.

Pick any closed-source operating system competing with Linux, and a best source for accounts of current development work on it.

Watch that source and Freshmeat for one month. Every day, count the number of release announcements on Freshmeat that you consider `original' work. Apply the same definition of `original' to announcements for that other OS and count them.

Thirty days later, total up both figures.

The day I wrote this, Freshmeat carried twenty-two release announcements, of which three appear they might push state of the art in some respect, This was a slow day for Freshmeat, but I will be astonished if any reader reports as many as three likely innovations **a month** in any closed-source channel.

[EGCS] We how have history on a project that, in several ways, may provide a more indicative test of the bazaar premise than fetchmail; EGCS, the Experimental GNU Compiler System.

This project was announced in mid-August of 1997 as a conscious attempt to apply the ideas in the early public versions of ``The Cathedral and the Bazaar''. The project founders felt that the development of GCC, the Gnu C Compiler, had been stagnating. For about twenty months afterwards, GCC and EGCS continued as parallel products -- both drawing from the same Internet developer population, both starting from the same GCC source base, both using pretty much the same Unix toolsets and development environment. The projects differed only in that EGCS consciously tried to apply the bazaar tactics I have previously described, while GCC retained a more cathedral-like organization with a closed developer group and infrequent releases.

This was about as close to a controlled experiment as one could ask for, and the results were dramatic. Within months, the EGCS versions had pulled substantially ahead in features; better optimization, better support for FORTRAN and C ++. Many people found the EGCS development snapshots to be more reliable than the most recent stable version of GCC, and major Linux distributions began to switch to EGCS.

In April of 1999, the Free Software Foundation (the official sponsors of GCC) dissolved the original GCC development group and officially handed control of the project to the the EGCS steering team.

[SP] Of course, Kropotkin's critique and Linus's Law raise some wider issues about the cybernetics of social organizations. Another folk theorem of software engineering suggests one of them; Conway's Law -- commonly stated as ``If you have four groups working on a compiler, you'll get a 4-pass compiler''. The original statement was more general: ``Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.'' We might put it more succinctly as ``The means determine the ends'', or even ``Process becomes product''.

It is accordingly worth noting that in the open-source community organizational form and function match on many levels. The network is everything and everywhere: not just the Internet, but the people doing the work form a distributed, loosely coupled, peer-to-peer network which provides multiple redundancy and degrades very gracefully. In both networks, each node is important only to the extent that other nodes want to cooperate with it.

The peer-to-peer part is essential to the community's astonishing productivity. The point Kropotkin was trying to make about power relationships is developed further by the `SNAFU Principle': ``True communication is possible only between equals, because inferiors are more consistently rewarded for telling their superiors pleasant lies than for telling the truth.'' Creative teamwork utterly depends on true communication and is thus very seriously hindered by the presence of power relationships. The open-source community, effectively free of such power relationships, is teaching us by contrast how dreadfully much they cost in bugs, in lowered productivity, and in lost opportunities.

Further, the SNAFU principle predicts in authoritarian organizations a progressive disconnect between decision-makers and reality, as more and more of the input to those who decide tends to become pleasant lies. The way this plays out in conventional software development is easy to see; there are strong incentives for the inferiors to hide, ignore, and minimize problems. When this process becomes product, software is a disaster.

Homesteading the Noosphere

1. An Introductory Contradiction

Anyone who watches the busy, tremendously productive world of Internet open-source software for a while is bound to notice an interesting contradiction between what open-source hackers say they believe and the way they actually behave -- between the official ideology of the open-source culture and its actual practice.

Cultures are adaptive machines. The open-source culture is a response to an identifiable set of drives and pressures. As usual, the culture's adaptation to its circumstances manifests both as conscious ideology and as implicit, unconscious or semi-conscious knowledge. And, as is not uncommon, the unconscious adaptations are partly at odds with the conscious ideology.

In this paper, I will dig around the roots of that contradiction, and use it to discover those drives and pressures. We will deduce some interesting things about the hacker culture and its customs. We will conclude by suggesting ways in which the culture's implicit knowledge can be leveraged better.

2. The Varieties of Hacker Ideology

The ideology of the Internet open-source culture (what hackers say they believe) is a fairly complex topic in itself. All members agree that open source (that is, software that is freely redistributable and can readily be evolved and modified to fit changing needs) is a good thing and worthy of significant and collective effort. This agreement effectively defines membership in the culture. However, the reasons individuals and various subcultures give for this belief vary considerably.

One degree of variation is zealotry; whether open source development is regarded merely as a convenient means to an end (good tools and fun toys and an interesting game to play) or as an end in itself.

A person of great zeal might say ``Free software is my life! I exist to create useful, beautiful programs and information resources, and then give them away.'' A person of moderate zeal might say ``Open source is a good thing, which I am willing to spend significant time helping happen''. A person of little zeal might say ``Yes, open source is OK sometimes. I play with it and respect people who build it''.

Another degree of variation is in hostility to commercial software and/or the companies perceived to dominate the commercial software market.

A very anticommercial person might say ``Commercial software is theft and hoarding. I write free software to end this evil.'' A moderately anticommercial person might say ``Commercial software in general is OK because programmers deserve to get paid, but companies that coast on shoddy products and throw their weight around are evil.'' An un-anticommercial person might say ``Commercial software is OK, I just use and/or write open-source software because I like it better''. (Nowadays, given the growth of the open-source part of the industry since the first public version of this paper, one might also hear ``Commercial software is fine, as long as I get the source or it does what I want it to do.'')

All nine of the attitudes implied by the cross-product of the above categories are represented in the open-source culture. The reason it is worthwhile to point out the distinctions is because they imply different agendas, and different adaptive and cooperative behaviors.

Historically, the most visible and best-organized part of the hacker culture has been both very zealous and very anticommercial. The Free Software Foundation founded by Richard M. Stallman (RMS) supported a great deal of open-source development from the early 1980s on, including tools like Emacs and GCC which are still basic to the Internet open-source world, and seem likely to remain so for the forseeable future.

For many years the FSF was the single most important focus of open-source hacking, producing a huge number of tools still critical to the culture. The FSF was also long the only sponsor of open source with an institutional identity visible to outside observers of the hacker culture. They effectively defined the term `free software', deliberately giving it a confrontational weight (which the newer label `open source' just as deliberately avoids).

Thus, perceptions of the hacker culture from both within and outside it tended to identify the culture with the FSF's zealous attitude and perceived anticommercial aims. RMS himself denies he is anticommercial, but his program has been so read by most people, including many of his most vocal partisans. The FSF's vigorous and explicit drive to ``Stamp Out Software Hoarding!'' became the closest thing to a hacker ideology, and RMS the closest thing to a leader of the hacker culture.

The FSF's license terms, the ``General Public License'' (GPL), expresses the FSF's attitudes. It is very widely used in the open-source world. North Carolina's Metalab (formerly Sunsite) is the largest and most popular software archive in the Linux world. In July 1997 about half the Sunsite software packages with explicit license terms used GPL.

But the FSF was never the only game in town. There was always a quieter, less confrontational and more market-friendly strain in the hacker culture. The pragmatists were loyal not so much to an ideology as to a group of engineering traditions founded on early open-source efforts which predated the FSF. These traditions included, most importantly, the intertwined technical cultures of Unix and the pre-commercial Internet.

The typical pragmatist attitude is only moderately anticommercial, and its major grievance against the corporate world is not `hoarding' per se. Rather it is that world's perverse refusal to adopt superior approaches incorporating Unix and open standards and open-source software. If the pragmatist hates anything, it is less likely to be `hoarders' in general than the current King Log of the software establishment; formerly IBM, now Microsoft.

To pragmatists, the GPL is important as a tool rather than an end in itself. Its main value is not as a weapon against `hoarding', but as a tool for encouraging software sharing and the growth of bazaar-mode development communities. The pragmatist values having good tools and toys more than he dislikes commercialism, and may use high-quality commercial software without ideological discomfort. At the same time, his open-source experience has taught him standards of technical quality that very little closed software can meet.

For many years, the pragmatist point of view expressed itself within the hacker culture mainly as a stubborn current of refusal to completely buy into the GPL in particular or the FSF's agenda in general. Through the 1980s and early 1990s, this attitude tended to be associated with fans of Berkeley Unix, users of the

BSD license, and the early efforts to build open-source Unixes from the BSD source base. These efforts, however, failed to build bazaar communities of significant size, and became seriously fragmented and ineffective.

Not until the Linux explosion of early 1993-1994 did pragmatism find a real power base. Although Linus Torvalds never made a point of opposing RMS, he set an example by looking benignly on the growth of a commercial Linux industry, by publicly endorsing the use of high-quality commercial software for specific tasks, and by gently deriding the more purist and fanatical elements in the culture.

A side effect of the rapid growth of Linux was the induction of a large number of new hackers for which Linux was their primary loyalty and the FSF's agenda primarily of historical interest. Though the newer wave of Linux hackers might describe the system as ``the choice of a GNU generation'', most tended to emulate Torvalds more than Stallman.

Increasingly it was the anticommercial purists who found themselves in a minority. How much things had changed would not become apparent until the Netscape announcement in February 1998 that it would distribute Navigator 5.0 in source. This excited more interest in `free software' within the corporate world. The subsequent call to the hacker culture to exploit this unprecedented opportunity and to re-label its product from `free software' to `open source' was met with a level of instant approval that surprised everybody involved.

In a reinforcing development, the pragmatist part of the culture was itself becoming polycentric by the mid-1990s. Other semi-independent communities with their own self-consciousness and charismatic leaders began to bud from the Unix/Internet root stock. Of these, the most important after Linux was the Perl culture under Larry Wall. Smaller, but still significant, were the traditions building up around John Osterhout's Tcl and Guido van Rossum's Python languages. All three of these communities expressed their ideological independence by devising their own, non-GPL licensing schemes.

3. Promiscuous Theory, Puritan Practice

Through all these changes, nevertheless, there remained a broad consensus theory of what `free software' or `open source' is. The clearest expression of this common theory can be found in the various open-source licenses, all of which have crucial common elements.

In 1997 these common elements were distilled into the Debian Free Software Guidelines, which became the Open Source Definition. Under the guidelines defined by the OSD, an open-source license must protect an unconditional right of any party to modify (and redistribute modified versions of) open-source software.

Thus, the implicit theory of the OSD (and OSD-conformant licenses such as the GPL, the BSD license, and Perl's Artistic License) is that anyone can hack anything. Nothing prevents half a dozen different people from taking any given open-source product (such as, say the Free Software Foundations's gcc C compiler), duplicating the sources, running off with them in different evolutionary directions, but all claiming to be *the* product.

In practice, however, such `forking' almost never happens. Splits in major projects have been rare, and always accompanied by re-labeling and a large volume of public self-justification. It is clear that, in such cases as the GNU Emacs/XEmacs split, or the gcc/egcs split, or the various fissionings of the BSD splinter groups, that the splitters felt they were going against a fairly powerful community norm [SP].

In fact (and in contradiction to the anyone-can-hack-anything consensus theory) the open-source culture has an elaborate but largely unadmitted set of ownership customs. These customs regulate who can modify software, the circumstances under which it can be modified, and (especially) who has the right to redistribute modified versions back to the community.

The taboos of a culture throw its norms into sharp relief. Therefore, it will be useful later on if we summarize some important ones here.

There is strong social pressure against forking projects. It does not happen except under plea of dire necessity, with much public self-justification, and with a renaming.

Distributing changes to a project without the cooperation of the moderators is frowned upon, except in special cases like essentially trivial porting fixes.

Removing a person's name from a project history, credits or maintainer list is absolutely *not done* without the person's explicit consent.

In the remainder of this paper, we shall examine these taboos and ownership customs in detail. We shall inquire not only into how they function but what they reveal about the underlying social dynamics and incentive structures of the open-source community.

4. Ownership and Open Source

What does `ownership' mean when property is infinitely reduplicable, highly malleable, and the surrounding culture has neither coercive power relationships nor material scarcity economics?

Actually, in the case of the open-source culture this is an easy question to answer. The owner(s) of a software project are those who have the exclusive right, recognized by the community at large, to *re-distribute modified versions*.

(In discussing `ownership' in this section I will use the singular, as though all projects are owned by some one person. It should be understood, however, that projects may be owned by groups. We shall examine the internal dynamics of such groups later in this paper.)

According to the standard open-source licenses, all parties are equals in the evolutionary game. But in practice there is a very well-recognized distinction between `official' patches, approved and integrated into the evolving software by the publicly recognized maintainers, and `rogue' patches by third parties. Rogue patches are unusual, and generally not trusted [RP].

That *public* redistribution is the fundamental issue is easy to establish. Custom encourages people to patch software for personal use when necessary. Custom is indifferent to people who redistribute modified versions within a closed user or development group. It is only when modifications are posted to the open-source community in general, to compete with the original, that ownership becomes an issue.

There are, in general, three ways to acquire ownership of an open-source project. One, the most obvious, is to found the project. When a project has had only one maintainer since its inception and the maintainer is still active, custom does not even permit a *question* as to who owns the project.

The second way is to have ownership of the project handed to you by the previous owner (this is sometimes known as `passing the baton'). It is well understood in the community that project owners have a duty to pass projects to competent successors when they are no longer willing or able to invest needed time in development or maintenance work.

It is significant that in the case of major projects, such transfers of control are generally announced with some fanfare. While it is unheard of for the open-source community at large to actually interfere in the owner's choice of succession, customary practice clearly incorporates a premise that public legitimacy is important.

For minor projects, it is generally sufficient for a change history included with the project distribution to note the change of ownership. The clear presumption is that if the former owner has not in fact voluntarily transferred control, he or she may reassert control with community backing by objecting publicly within a reasonable period of time.

The third way to acquire ownership of a project is to observe that it needs work and the owner has disappeared or lost interest. If you want to do this, it is your responsibility to make the effort to find the owner. If you don't succeed, then you may announce in a relevant place (such as a Usenet newsgroup dedicated to the application area) that the project appears to be orphaned, and that you are considering taking responsibility for it.

Custom demands that you allow some time to pass before following up with an announcement that you have declared yourself the new owner. In this interval, if someone else announces that they have been actually working on the project, their claim trumps yours. It is considered good form to give public notice of your intentions more than once. More points for good form if you announce in many relevant forums (related newsgroups, mailing lists); and still more if you show patience in waiting for replies. In general, the more visible effort you make to allow the previous owner or other claimants to respond, the better your claim if no response is forthcoming.

If you have gone through this process in sight of the project's user community, and there are no objections, then you may claim ownership of the orphaned project and so note in its history file. This, however, is less secure than being passed the baton, and you cannot expect to be considered fully legitimate until you have made substantial improvements in the sight of the user community.

I have observed these customs in action for twenty years, going back to the pre-FSF ancient history of open-source software. They have several very interesting features. One of the most interesting is that most hackers have followed them without being fully aware of doing so. Indeed, the above may be the first conscious and reasonably complete summary ever to have been written down.

Another is that, for unconscious customs, they have been followed with remarkable (even astonishing) consistency. I have observed the evolution of literally hundreds of open-source projects, and I can still count the number of significant violations I have observed or heard about on my fingers.

Yet a third interesting feature is that as these customs have evolved over time, they have done so in a consistent direction. That direction has been to encourage more public accountability, more public notice, and more care about preserving the credits and change histories of projects in ways which (among other things) establish the legitimacy of the present owners.

These features suggest that the customs are not accidental, but are products of some kind of implicit agenda or generative pattern in the open-source culture that is utterly fundamental to the way it operates.

An early respondent pointed out that contrasting the Internet hacker culture with the cracker/pirate culture (the ``warez d00dz'' centered around game-cracking and pirate bulletin-board systems) illuminates the generative patterns of both rather well. We'll return to the d00dz for contrast later in the paper.

5. Locke and Land Title

To understand this generative pattern, it helps to notice a historical analogy for these customs that is far outside the domain of hackers' usual concerns. As students of legal history and political philosophy may recognize, the theory of property they imply is virtually identical to the Anglo-American common-law theory of land tenure!

In this theory, there are three ways to acquire ownership of land.

On a frontier, where land exists that has never had an owner, one can acquire ownership by *homesteading*, mixing one's labor with the unowned land, fencing it, and defending one's title.

The usual means of transfer in settled areas is *transfer of title* -- that is, receiving the deed from the previous owner. In this theory, the concept of `chain of title' is important. The ideal proof of ownership is a chain of deeds and transfers extending back to when the land was originally homesteaded.

Finally, the common-law theory recognizes that land title may be lost or abandoned (for example, if the owner dies without heirs, or the records needed to establish chain of title to vacant land are gone). A piece of land that has become derelict in this way may be claimed by *adverse possession* – one moves in, improves it, and defends title as if homesteading.

This theory, like hacker customs, evolved organically in a context where central authority was weak or nonexistent. It developed over a period of a thousand years from Norse and Germanic tribal law. Because it was systematized and rationalized in the early modern era by the English political philosopher John Locke, it is sometimes referred to as the `Lockean' theory of property.

Logically similar theories have tended to evolve wherever property has high economic or survival value and no single authority is powerful enough to force central allocation of scarce goods. This is true even in the hunter-gatherer cultures that are sometimes romantically thought to have no concept of `property'. For example, in the traditions of the !Kung San bushmen of the Kgalagadi (formerly `Kalahari') Desert, there is no ownership of hunting grounds. But there *is* ownership of water-holes and springs under a theory recognizably akin to Locke's.

The !Kung San example is instructive, because it shows that Lockean property customs arise only where the expected return from the resource exceeds the expected cost of defending it. Hunting grounds are not property because the return from hunting is highly unpredictable and variable, and (although highly prized) not a necessity for day-to-day survival. Waterholes, on the other hand, are vital to survival and small enough to defend.

The `noosphere' of this essay's title is the territory of ideas, the space of all possible thoughts [N]. What we see implied in hacker ownership customs is a Lockean theory of property rights in one subset of the noosphere, the space of all programs. Hence `homesteading the noosphere', which is what every founder of a new open-source project does.

Faré Rideau <fare@tunes.org> correctly points out that hackers do not exactly operate in the territory of pure ideas. He asserts that what hackers own is *programming projects* -- intensional focus points of material labor (development, service, etc), to which are associated things like reputation, trustworthiness, etc. He therefore asserts that the space spanned by hacker projects, is *not* the noosphere but a sort of dual of it, the space of noosphere-exploring program projects. (With an apologetic nod to the astrophysicists out there, it would be etymologically correct to call this dual space the `ergosphere' or `sphere of work'.)

In practice, the distinction between noosphere and ergosphere is not important for the purposes of this paper. It is dubious whether the `noosphere' in the pure sense Faré insists on can be said to exist in any meaningful way; one would almost have to be a Platonist philosopher to believe in it. And the distinction between noosphere and ergosphere is only of *practical* importance if one wishes to assert that ideas (the elements of the noosphere) cannot be owned, but their instantiations as projects can. This question leads to issues in the theory of intellectual property which are beyond the scope of this paper (but see [DF]).

To avoid confusion, however, it is important to note that neither the noosphere nor the ergosphere is the same as the totality of virtual locations in electronic media that is sometimes (to the disgust of most hackers) called `cyberspace'. Property there is regulated by completely different rules that are closer to those of the material substratum-- essentially, he who owns the media and machines on which a part of `cyberspace' is hosted owns that piece of cyberspace as a result.

The Lockean logic of custom suggests strongly that open-source hackers observe the customs they do in order to defend some kind of expected return from their effort. The return must be more significant than the effort of homesteading projects, the cost of maintaining version histories that document `chain of title', and the time cost of doing public notifications and a waiting period before taking adverse possession of an orphaned project.

Furthermore, the `yield' from open source must be something more than simply the use of the software, something else that would be compromised or diluted by forking. If use were the only issue, there would be no taboo against forking, and open-source ownership would not resemble land tenure at all. In fact, this alternate world (where use is the only yield, and forking is unproblematic) is the one implied by existing open-source licenses.

We can eliminate some candidate kinds of yield right away. Because you can't coerce effectively over a network connection, seeking power is right out. Likewise, the open-source culture doesn't have anything much resembling money or an internal scarcity economy, so hackers cannot be pursuing anything very closely analogous to material wealth (e.g. the accumulation of scarcity tokens).

There is one way that open-source activity can help people become wealthier, however -- a way that provides a valuable clue to what actually motivates it. Occasionally, the reputation one gains in the hacker culture can spill over into the real world in economically significant ways. It can get you a better job offer, or a consulting contract, or a book deal.

This kind of side effect, however, is at best rare and marginal for most hackers; far too much so to make it convincing as a sole explanation, even if we ignore the repeated protestations by hackers that they're doing what they do not for money but out of idealism or love.

However, the way such economic side-effects are mediated is worth examination. Below we'll see that an understanding of the dynamics of reputation within the open-source culture *itself* has considerable explanatory power.

6. The Hacker Milieu as Gift Culture

To understand the role of reputation in the open-source culture, it is helpful to move from history further into anthropology and economics, and examine the difference between *exchange cultures* and *gift cultures*.

Human beings have an innate drive to compete for social status; it's wired in by our evolutionary history. For the 90% of that history that ran before the invention of agriculture, our ancestors lived in small nomadic hunting-gathering bands. High-status individuals (those most effective at informing coalitions and persuading others to cooperate with them) got the healthiest mates and access to the best food. This drive for status expresses itself in different ways, depending largely on the degree of scarcity of survival goods.

Most ways humans have of organizing are adaptations to scarcity and want. Each way carries with it different ways of gaining social status.

The simplest way is the *command hierarchy*. In command hierarchies, allocation of scarce goods is done by one central authority and backed up by force. Command hierarchies scale very poorly [Mal]; they become increasingly brutal and inefficient as they get larger. For this reason, command hierarchies above the size of an extended family are almost always parasites on a larger economy of a different type. In command hierarchies, social status is primarily determined by access to coercive power.

Our society is predominantly an *exchange economy*. This is a sophisticated adaptation to scarcity that, unlike the command model, scales quite well. Allocation of scarce goods is done in a decentralized way through trade and voluntary cooperation (and in fact, the dominating effect of competitive desire is to produce cooperative behavior). In an exchange economy, social status is primarily determined by having control of things (not necessarily material things) to use or trade.

Most people have implicit mental models for both of the above, and how they interact with each other. Government, the military, and organized crime (for example) are command hierarchies parasitic on the broader exchange economy we call `the free market'. There's a third model, however, that is radically different from either and not generally recognized except by anthropologists; the *gift culture*.

Gift cultures are adaptations not to scarcity but to abundance. They arise in populations that do not have significant material-scarcity problems with survival goods. We can observe gift cultures in action among aboriginal cultures living in ecozones with mild climates and abundant food. We can also observe them in certain strata of our own society, especially in show business and among the very wealthy.

Abundance makes command relationships difficult to sustain and exchange relationships an almost pointless game. In gift cultures, social status is determined not by what you control but by *what you give away*.

Thus the Kwakiutl chieftain's potlach party. Thus the multi-millionaire's elaborate and usually public acts of philanthropy. And thus the hacker's long hours of effort to produce high-quality open-source code.

For examined in this way, it is quite clear that the society of open-source hackers is in fact a gift culture. Within it, there is no serious shortage of the `survival necessities' -- disk space, network bandwidth, computing power. Software is freely shared. This abundance creates a situation in which the only available measure of competitive success is reputation among one's peers.

This observation is not in itself entirely sufficient to explain the observed features of hacker culture, however. The crackers and warez d00dz have a gift culture that thrives in the same (electronic) media as that of the hackers, but their behavior is very different. The group mentality in their culture is much stronger and more exclusive than among hackers. They hoard secrets rather than sharing them; one is much more likely to find cracker groups distributing sourceless executables that crack software than tips that give away how they did it. (For an inside perspective on this behavior, see [LW]).

What this shows, in case it wasn't obvious, is that there is more than one way to run a gift culture. History and values matter. I have summarized the history of the hacker culture in *A Brief History of Hackerdom* ; the ways in which it shaped present behavior are not mysterious. Hackers have defined their culture by a set of choices about the *form* which their competition will take. It is that form which we will examine in the remainder of this paper.

7. The Joy of Hacking

In making this `reputation game' analysis, by the way, I do not mean to devalue or ignore the pure artistic satisfaction of designing beautiful software and making it work. We all experience this kind of satisfaction and thrive on it. People for whom it is not a significant motivation never become hackers in the first place, just as people who don't love music never become composers.

So perhaps we should consider another model of hacker behavior in which the pure joy of craftsmanship is the primary motivation. This `craftsmanship' model would have to explain hacker custom as a way of maximizing both the opportunities for craftsmanship and the quality of the results. Does this conflict with or suggest different results than the `reputation game' model?

Not really. In examining the `craftsmanship' model, we come back to the same problems that constrain hackerdom to operate like a gift culture. How can one maximize quality if there is no metric for quality? If scarcity economics doesn't operate, what metrics are available besides peer evaluation? It appears that any craftsmanship culture ultimately must structure itself through a reputation game -- and, in fact, we can observe exactly this dynamic in many historical craftsmanship cultures from the medieval guilds onwards.

In one important respect, the `craftsmanship' model is weaker than the `gift culture' model; by itself, it doesn't help explain the contradiction we began this paper with.

Finally, the `craftsmanship' motivation itself may not be psychologically as far removed from the reputation game as we might like to assume. Imagine your beautiful program locked up in a drawer and never used again. Now imagine it being used effectively and with pleasure by many people. Which dream gives you satisfaction?

Nevertheless, we'll keep an eye on the craftsmanship model. It is intuitively appealing to many hackers, and explains some aspects of individual behavior well enough [HT].

After I published the first version of this paper on the Internet, an anonymous respondent commented: ``You may not work to get reputation, but the reputation is a real payment with consequences if you do the job well.'' This is a subtle and important point. The reputation incentives continue to operate whether or not a craftsman is aware of them; thus, ultimately, whether or not a hacker understands his own behavior as part of the reputation game, his behavior will be shaped by that game.

Other respondents related peer-esteem rewards and the joy of hacking to the levels above subsistence needs in Abraham Maslow's well-known `hierarchy of values' model of human motivation [MH]. On this view, the joy of hacking is a self-actualization or transcendence need which will not be consistently expressed until lower-level needs (including those for physical security and for `belongingness' or peer esteem) have been at least minimally satisfied. Thus, the reputation game may be critical in providing a social context within which the joy of hacking can in fact *become* the individual's primary motive.

8. The Many Faces of Reputation

There are reasons general to every gift culture why peer repute (prestige) is worth playing for:

First and most obviously, good reputation among one's peers is a primary reward. We're wired to experience it that way for evolutionary reasons touched on earlier. (Many people learn to redirect their drive for prestige into various sublimations that have no obvious connection to a visible peer group, such as ``honor'', ``ethical integrity'', ``piety'' etc.; this does not change the underlying mechanism.)

Secondly, prestige is a good way (and in a pure gift economy, the *only* way) to attract attention and cooperation from others. If one is well known for generosity, intelligence, fair dealing, leadership ability, or other good qualities, it becomes much easier to persuade other people that they will gain by association with you.

Thirdly, if your gift economy is in contact with or intertwined with an exchange economy or a command hierarchy, your reputation may spill over and earn you higher status there.

Beyond these general reasons, the peculiar conditions of the hacker culture make prestige even more valuable than it would be in a `real world' gift culture.

The main `peculiar condition' is that the artifacts one gives away (or, interpreted another way, are the visible sign of one's gift of energy and time) are very complex. Their value is nowhere near as obvious as that of material gifts or exchange-economy money. It is much harder to objectively distinguish a fine gift from a poor one. Accordingly, the success of a giver's bid for status is delicately dependent on the critical judgement of peers.

Another peculiarity is the relative purity of the open-source culture. Most gift cultures are compromised -- either by exchange-economy relationships such as trade in luxury goods, or by command-economy relationships such as family or clan groupings. No significant analogues of these exist in the open-source culture; thus, ways of gaining status other than by peer repute are virtually absent.

9. Ownership Rights and Reputation Incentives

We are now in a position to pull together the previous analyses into a coherent account of hacker ownership customs. We

understand the yield from homesteading the noosphere now; it is peer repute in the gift culture of hackers, with all the secondary gains and side-effects that implies.

From this understanding, we can analyze the Lockean property customs of hackerdom as a means of *maximizing reputation incentives*; of ensuring that peer credit goes where it is due and does not go where it is not due.

The three taboos we observed above make perfect sense under this analysis. One's reputation can suffer unfairly if someone else misappropriates or mangles one's work; these taboos (and related customs) attempt to prevent this from happening. (Or, to put it more pragmatically, hackers generally refrain from forking or rogue-patching others projects in order to be able to deny legitimacy to the same behavior practiced against themselves.)

Forking projects is bad because it exposes pre-fork contributors to a reputation risk they can only control by being active in both child projects simultaneously after the fork. (This would generally be too confusing or difficult to be practical.)

Distributing rogue patches (or, much worse, rogue binaries) exposes the owners to an unfair reputation risk. Even if the official code is perfect, the owners will catch flak from bugs in the patches (but see [RP]).

Surreptitiously filing someone's name off a project is, in cultural context, one of the ultimate crimes. It steals the victim's gift to be presented as the thief's own.

Of course, forking a project or distributing rogue patches for it also directly attacks the reputation of the original developer's group. If I fork or rogue-patch your project, I am saying: "you made a wrong decision [by failing to take the project where I am taking it]"; and Anyone who uses my forked variation is endorsing this challenge. But this in itself would be a fair challenge, albeit extreme; it's the sharpest end of peer review. It's therefore not sufficient in itself to account for the taboos, though it doubtless contributes force to them.

All three of these taboo behaviors inflict global harm on the open-source community as well as local harm on the victim(s). Implicitly they damage the entire community by decreasing each potential contributor's perceived likelihood that gift/productive behavior will be rewarded.

It's important to note that there are alternate candidate explanations for two of these three taboos.

First, hackers often explain their antipathy to forking projects by bemoaning the wasteful duplication of work it would imply as the child products evolved in more-or-less parallel into the future. They may also observe that forking tends to split the co-developer community, leaving both child projects with fewer brains to work with than the parent.

A respondent has pointed out that it is unusual for more than one offspring of a fork to survive with significant `market share' into the long term. This strengthens the incentives for all parties to cooperate and avoid forking, because it's hard to know in advance who will be on the losing side and see a lot of their work either disappear entirely or languish in obscurity.

Dislike of rogue patches is often explained by observing that they can complicate bug-tracking enormously, and inflict work on maintainers who have quite enough to do catching their *own* mistakes.

There is considerable truth to these explanations, and they certainly do their bit to reinforce the Lockean logic of ownership. But while intellectually attractive, they fail to explain why so much emotion and territoriality gets displayed on the infrequent occasions that the taboos get bent or broken – not just by the injured parties, but by bystanders and observers who often react quite harshly. Cold-blooded concerns about duplication of work and maintainance hassles simply do not sufficiently explain the observed behavior.

Then, too, there is the third taboo. It's hard to see how anything but the reputation-game analysis can explain this. The fact that this taboo is seldom analyzed much more deeply than ``It wouldn't be fair'' is revealing in its own way, as we shall see in the next section.

10. The Problem of Ego

At the beginning of the paper I mentioned that the unconscious adaptive knowledge of a culture is often at odds with its conscious ideology. We've seen one major example of this already in the fact that Lockean ownership customs have been widely followed despite the fact that they violate the stated intent of the standard licenses.

I have observed another interesting example of this phenomenon when discussing the reputation-game analysis with hackers. This is that many hackers resisted the analysis and showed a strong reluctance to admit that their behavior was motivated by a desire for peer repute or, as I incautiously labeled it at the time, `ego satisfaction'.

This illustrates an interesting point about the hacker culture. It consciously distrusts and despises egotism and ego-based motivations; Self-promotion tends to be mercilessly criticized, even when the community might appear to have something to gain from it. So much so, in fact, that the culture's `big men' and tribal elders are required to talk softly and humorously deprecate themselves at every turn in order to maintain their status. How this attitude meshes with an incentive structure that apparently runs almost entirely on ego cries out for explanation.

A large part of it, certainly, stems from the generally negative Europo-American attitude towards `ego'. The cultural matrix of most hackers teaches them that desiring ego satisfaction is a bad (or at least immature) motivation; that ego is at best an eccentricity tolerable only in prima-donnas and often an actual sign of mental pathology. Only sublimated and disguised forms like ``peer repute'', ``self-esteem'', ``professionalism'' or ``pride of accomplishment'' are generally acceptable.

I could write an entire other essay on the unhealthy roots of this part of our cultural inheritance, and the astonishing amount of self-deceptive harm we do by believing (against all the evidence of psychology and behavior) that we ever have truly `selfless' motives. Perhaps I would, if Friedrich Wilhelm Nietzsche and Ayn Rand had not already done an entirely competent job (whatever their other failings) of deconstructing `altruism' into unacknowledged kinds of self-interest.

But I am not doing moral philosophy or psychology here, so I will simply observe one minor kind of harm done by the belief that ego is evil, which is this: it has made it emotionally difficult for many hackers to consciously understand the social dynamics of their own culture!

But we are not quite done with this line of investigation. The surrounding culture's taboo against visibly ego-driven behavior is so much intensified in the hacker (sub)culture that one must suspect it of having some sort of special adaptive function for hackers. Certainly the taboo is weaker (or nonexistent) among many other gift cultures, such as the peer cultures of theater people or the very wealthy!

11. The Value of Humility

Having established that prestige is central to the hacker culture's reward mechanisms, we now need to understand why it has seemed so important that this fact remain semi-covert and largely unadmitted.

The contrast with the pirate culture is instructive. In that culture, status-seeking behavior is overt and even blatant. These crackers seek acclaim for releasing ``zero-day warez'' (cracked software redistributed on the day of the original uncracked version's release) but are closemouthed about how they do it. These magicians don't like to give away their tricks. And, as a result, the knowledge base of the cracker culture as a whole increases only slowly.

In the hacker community, by contrast, one's work is one's statement. There's a very strict meritocracy (the best craftsmanship wins) and there's a strong ethos that quality should (indeed *must*) be left to speak for itself. The best brag is code that ``just works'', and that any competent programmer can see is good stuff. Thus, the hacker culture's knowledge base increases rapidly.

The taboo against ego-driven posturing therefore increases productivity. But that's a second-order effect; what is being directly protected here is the quality of the information in the community's peer-evaluation system. That is, boasting or self-importance is suppressed because it behaves like noise tending to corrupt the vital signals from experiments in creative and cooperative behavior.

For very similar reasons, attacking the author rather than the code is not done. There is an interesting subtlety here that reinforces the point; hackers feel very free to flame each other over ideological and personal differences, but it is unheard of for any hacker to publicly attack another's competence at technical work (even private criticism is unusual and tends to be muted in tone). Bug-hunting and criticism are always project-labeled, not person-labeled.

Furthermore, past bugs are not automatically held against a developer; the fact that a bug has been fixed is generally considered more important than the fact that one used to be there. As one respondent observed, one can gain status by fixing `Emacs bugs', but not by fixing `Richard Stallman's bugs' -- and it would be considered extremely bad form to criticize Stallman for *old* Emacs bugs that have since been fixed.

This makes an interesting contrast with many parts of academia, in which trashing putatively defective work by others is an important mode of gaining reputation. In the hacker culture, such behavior is rather heavily tabooed -- so heavily, in fact, that the absence of such behavior did not present itself to me as a datum until that one respondent with an unusual perspective pointed it out nearly a full year after this paper was first published!

The taboo against attacks on competence (not shared with academia) is even more revealing than the (shared) taboo on posturing, because we can relate it to a difference between academia and hackerdom in their communications and support structures.

The hacker culture's medium of gifting is intangible, its communications channels are poor at expressing emotional nuance, and face-to-face contact among its members is the exception rather than the rule. This gives it a lower tolerance of noise than most other gift cultures, and goes a long way to explain the taboo against attacks on competence. Any significant incidence of flames over hackers' competence would intolerably disrupt the culture's reputation scoreboard.

The same vulnerability to noise goes for to explain the example in public humility required of the hacker community's tribal elders. They must be seen to be free of boast and posturing so the taboo against dangerous noise will hold. [DC]

Talking softly is also functional if one aspires to be a maintainer of a successful project; one must convince the community that one has good judgement, because most of the maintainer's job is going to be judging other people's code. Who would be inclined to contribute work to someone who clearly can't judge the quality of their own code, or whose behavior suggests they will attempt to unfairly hog the reputation return from the project? Potential contributors want project leaders with enough humility and class be able to to say, when objectively appropriate, ``Yes, that does work better than my version, I'll use it'' -- and to give credit where credit is due.

Yet another reason for humble behavior is that in the open source world, you seldom want to give the impression that a project is `done'. This might lead a potential contributor not to feel needed. The way to maximize your leverage is to be humble about the state of the program. If one does one's bragging through the code, and then says ``Well shucks, it doesn't do x, y, and z, so it can't be that good'', patches for x, y, and z will often swiftly follow.

Finally, I have personally observed that the self-deprecating behavior of some leading hackers reflects a real (and not unjustified) fear of becoming the object of a personality cult. Linus Torvalds and Larry Wall both provide clear and numerous examples of such avoidance behavior. Once, on a dinner expedition with Larry Wall, I joked ``You're the alpha hacker here -- you get to pick the restaurant''. He flinched audibly. And rightly so; failing to distinguish their shared values from the personalities of their leaders has ruined a good many voluntary communities, a pattern of which Larry and Linus cannot fail to be fully aware. On the other hand, most hackers would love to have Larry's problem, if they could but bring themselves to admit it.

12. Global Implications of the Reputation-Game Model

The reputation-game analysis has some more implications that may not be immediately obvious. Many of these derive from the fact that one gains more prestige from founding a successful project than from cooperating in an existing one. One also gains more from projects which are strikingly innovative, as opposed to being `me, too' incremental improvements on software that already exists. On the other hand, software that nobody but the author understands or has a need for is a non-starter in the reputation game, and it's often easier to attract good notice by contributing to an existing project than it is to get people to notice a new one. Finally, it's much harder to compete with an already successful project than it is to fill an empty niche.

Thus, there's an optimum distance from one's neighbors (the most similar competing projects). Too close and one's product will be a ``me, too!'' of limited value, a poor gift (one would be better off contributing to an existing project). Too far away, and nobody will be able to use, understand, or perceive the relevance of one's effort (again, a poor gift). This creates a pattern of homesteading in the noosphere that rather resembles that of settlers spreading into a physical frontier -- not random, but like a diffusion-limited fractal. Projects tend to get started to fill

functional gaps near the frontier (see [NO] for further discussion of the lure of novelty).

Some very successful projects become `category killers'; nobody wants to homestead anywhere near them because competing against the established base for the attention of hackers would be too hard. People who might otherwise found their own distinct efforts end up, instead, adding extensions for these big, successful projects. The classic `category killer' example is GNU Emacs; its variants fill the ecological niche for a fully-programmable editor so completely that no competitor has gotten much beyond the one-man project stage since the early 1980s. Instead, people write Emacs modes.

Globally, these two tendencies (gap-filling and category-killers) have driven a broadly predictable trend in project starts over time. In the 1970s most of the open source that existed was toys and demos. In the 1980s the push was in development and Internet tools. In the 1990s the action shifted to operating systems. In each case, a new and more difficult level of problems was attacked when the possibilities of the previous one had been nearly exhausted.

This trend has interesting implications for the near future. In early 1998, Linux looks very much like a category-killer for the niche `open-source operating systems' -- people who might otherwise write competing operating systems are now writing Linux device drivers and extensions instead. And most of the lower-level tools the culture ever imagined having as open-source already exist. What's left?

Applications. As the year 2000 approaches, it seems safe to predict that open-source development effort will increasingly shift towards the last virgin territory -- programs for non-techies. A clear early indicator is the development of GIMP, the Photoshop-like image workshop that is open source's first major application with the kind of end-user-friendly GUI interface considered *de rigueur* in commercial applications for the last decade. Another is the amount of buzz surrounding application-toolkit projects like KDE and GNOME.

A respondent to this paper has pointed out that the homesteading analogy also explains why hackers react with such visceral anger to Microsoft's ``embrace and extend'' policy of complexifying and then closing up Internet protocols. The hacker culture can coexist with most closed software; the existence of Adobe Photoshop, for example, does not make the territory near GIMP (its open-source equivalent) significantly less attractive. But when Microsoft succeeds at de-commoditizing [HD] a protocol so that only Microsoft's own programmers can write software for it, they do not merely harm customers by extending their monopoly. They also reduce the amount and quality of noosphere available for hackers to homestead and cultivate. No wonder hackers often refer to Microsoft's strategy as ``protocol pollution''; they are reacting exactly like farmers watching someone poison the river they water their crops with!

Finally, the reputation-game analysis explains the oft-cited dictum that you do not become a hacker by calling yourself a hacker -- you become a hacker when *other hackers* call you a hacker. A `hacker', considered in this light, is somebody who has shown (by contributing gifts) that he or she both has technical ability and understands how the reputation game works. This judgement is mostly one of awareness and acculturation, and can only be delivered by those already well inside the culture.

13. How Fine a Gift?

There are consistent patterns in the way the hacker culture values contributions and returns peer esteem for them. It's not hard to observe the following rules:

1. If it doesn't work as well as I have been led to expect it will, it's no good -- no matter how clever and original it is.

Note the `led to expect'. This rule is not a demand for perfection; beta and experimental software is allowed to have bugs. It's a demand that the user be able to accurately estimate risks from the stage of the project and the developers' representations about it.

This rule underlies the fact that open-source software tends to stay in beta for a long time, and not get even a 1.0 version number until the developers are very sure it will not hand out a lot of nasty surprises. In the closed-source world, Version 1.0 means ``Don't touch this if you're prudent.''; in the open-source world it reads something more like ``The developers are willing to bet their reputations on this.''

2. Work that extends the noosphere is better than work that duplicates an existing piece of functional territory.

The naive way to put this would have been: *Original work is better than duplicating the functions of existing software.* But it's not actually quite that simple. Duplicating the functions of existing *closed* software counts as highly as original work if by doing so you break open a closed protocol or format and make that territory newly available.

Thus, for example, one of the highest-prestige projects in the present open-source world is Samba -- the code that allows Unix machines to act as clients or servers for Microsoft's proprietary SMB file-sharing protocol. There is very little creative work to be done here; it's mostly an issue of getting the reverse-engineered details right. Nevertheless, the members of the Samba group are perceived as heroes because they neutralize a Microsoft effort to lock in whole user populations and cordon off a big section of the noosphere.

3. Work that makes it into a major distribution is better than work that doesn't. Work carried in all major distributions is most prestigious.

The major distributions include not just the big Linux distributions like Red Hat, Debian, Caldera, and S.u.S.E., but other collections that are understood to have reputations of their own to maintain and thus implicitly certify quality -- like BSD distributions or the Free Software Foundation source collection.

4. Utilization is the sincerest form of flattery -- and category killers are better than also-rans.

Trusting the judgment of others is basic to the peer-review process. It's necessary because nobody has time to review all possible alternatives. So work used by lots of people is considered better than work used by a few,

To have done work so good that nobody cares to use the alternatives any more is therefore to have earned huge prestige. The most possible peer esteem comes from having done widely popular, category-killing original work that is carried by all major distributions. People who have pulled this off more than once are half-seriously referred to as `demigods'.

5. Continued devotion to hard, boring work (like debugging, or writing documentation) is more praiseworthy than cherrypicking the fun and easy hacks.

This norm is how the community rewards necessary tasks that hackers would not naturally incline towards. It is to some extent contradicted by:

6. Nontrivial extensions of function are better than low-level patches and debugging.

The way this seems to work is that on a one-shot basis, adding a feature is likely to get more reward than fixing a bug -- unless the bug is exceptionally nasty or obscure, such that nailing it is itself a demonstration of unusual skill and cleverness. But when these behaviors are extended over time, a person with a long history of paying attention to and nailing even ordinary bugs may well rank someone who has spent a similar amount of effort adding easy features.

A respondent has pointed out that these rules interact in interesting ways and do not necessarily reward highest possible utility all the time. Ask a hacker whether he's likely to become better known for a brand new tool of his own or for extensions to someone else's and the answer ``new tool'' will not be in doubt. But ask about

(a) a brand new tool which is only used a few times a day invisibly by the OS but which rapidly becomes a category killer

versus

(b) several extensions to an existing tool which are neither especially novel nor category-killers, but are daily used and daily visible to a huge number of users

and you are likely to get some hesitation before the hacker settles on (a). These alternatives are about evenly stacked.

Said respondent gave this question point for me by adding ``Case (a) is fetchmail; case (b) is your many Emacs extensions, like vc.el and gud.el.'' And indeed he is correct; I am more likely to be tagged `the author of fetchmail' than `author of a boatload of Emacs modes', even though the latter probably have had higher total utility over time.

What may be going on here is simply that work with a novel `brand identity' gets more notice than work aggregated to an existing `brand'. Elucidation of these rules, and what they tell us about the hacker culture's scoreboarding system, would make a good topic for further investigation.

14. Noospheric Property and the Ethology of Territory

To understand the causes and consequences of Lockean property customs, it will help us to look at them from yet another angle; that of animal ethology, specifically the ethology of territory.

Property is an abstraction of animal territoriality, which evolved as a way of reducing intra-species violence. By marking his bounds, and respecting the bounds of others, a wolf diminishes his chances of being in a fight that could weaken or kill him and make him less reproductively successful. Similarly, the function of property in human societies is to prevent inter-human conflict by setting bounds that clearly separate peaceful behavior from aggression.

It is fashionable in some circles to describe human property as an arbitrary social convention, but this is dead wrong. Anybody who has ever owned a dog who barked when strangers came near its owner's property has experienced the essential continuity between animal territoriality and human property. Our domesticated cousins of the wolf know, instinctively, that property is no mere social convention or game, but a critically important evolved mechanism for the avoidance of violence. (This makes them smarter than a good many human political theorists.)

Claiming property (like marking territory) is a performative act, a way of declaring what boundaries will be defended. Community support of property claims is a way to minimize friction and maximize cooperative behavior. These things remain true even when the ``property claim'' is much more abstract than a fence or a dog's bark, even when it's just the statement of the project maintainer's name in a README file. It's still an abstraction of territoriality, and (like other forms of property) based in territorial instincts evolved to assist conflict resolution.

This ethological analysis may at first seem very abstract and difficult to relate to actual hacker behavior. But it has some important consequences. One is in explaining the popularity of World Wide Web sites, and especially why open-source projects with websites seem so much more `real' and substantial than those without them.

Considered objectively, this seems hard to explain. Compared to the effort involved in originating and maintaining even a small program, a web page is easy, so it's hard to consider a web page evidence of substance or unusual effort.

Nor are the functional characteristics of the Web itself sufficient explanation. The communication functions of a web page can be as well or better served by a combination of an FTP site, a mailing list, and Usenet postings. In fact it's quite unusual for a project's routine communications to be done over the Web rather than via a mailing list or newsgroup. Why, then, the popularity of Web sites as project homes?

The metaphor implicit in the term `home page' provides an important clue. While founding an open-source project is a territorial claim in the noosphere (and customarily recognized as such) it is not a terribly compelling one on the psychological level. Software, after all, has no natural location and is instantly reduplicable. It's assimilable to our instinctive notions of `territory' and `property', but only after some effort.

A project home page concretizes an abstract homesteading in the space of possible programs by expressing it as `home' territory in the more spatially-organized realm of the World Wide Web. Descending from the noosphere to `cyberspace' doesn't get us all the way to the real world of fences and barking dogs yet, but it does hook the abstract property claim more securely to our instinctive wiring about territory. And this is why projects with web pages seem more `real'.

This point is much strengthened by hyperlinks and the existence of good search engines. A project with a web page is much more likely to be noticed by somebody exploring its neighborhood in the noosphere; others will link to it, searches will find it. A web page is therefore a better advertisement, a more effective performative act, a stronger claim on territory.

This ethological analysis also encourages us to look more closely at mechanisms for handling conflict in the open-source culture. It leads us to expect that, in addition to maximizing reputation incentives, ownership customs should also have a role in preventing and resolving conflicts.

15. Causes of Conflict

In conflicts over open-source software we can identify four major issues:

Who gets to make binding decisions about a project?

Who gets credit or blame for what?

How to reduce duplication of effort and prevent rogue versions from complicating bug tracking?

What is the Right Thing, technically speaking?

If we take a second look at the ``What is the Right Thing'' issue, however, it tends to vanish. For any such question, either there is an objective way to decide it accepted by all parties or there isn't. If there is, game over and everybody wins. If there isn't, it reduces to ``who decides?''.

Accordingly, the three problems a conflict-resolution theory has to resolve about a project are (A) where the buck stops on design decisions, (B) how to decide which contributors are credited and how, and (C) how to keep a project group and product from fissioning into multiple branches.

The role of ownership customs in resolving issues (A) and (C) is clear. Custom affirms that the owners of the project make the binding decisions. We have previously observed that custom also exerts heavy pressure against dilution of ownership by forking.

It's instructive to notice that these customs make sense even if one forgets the reputation game and examines them from within a pure `craftmanship' model of the hacker culture. In this view these customs have less to do with the dilution of reputation incentives than with protecting a craftsman's right to execute his vision in his chosen way.

The craftsmanship model is not, however, sufficient to explain hacker customs about issue (B), who gets credit for what (because a pure craftsman, one unconcerned with the reputation game, would have no motive to care). To analyze these, we need to take the Lockean theory one step further and examine conflicts and the operation of property rights *within* projects as well as *between* them.

16. Project Structures and Ownership

The trivial case is that in which the project has a single owner/maintainer. In that case there is no possible conflict. The owner makes all decisions and collects all credit and blame. The only possible conflicts are over succession issues -- who gets to be the new owner if the old one disappears or loses interest. The community also has an interest, under issue (C), in preventing forking. These interests are expressed by a cultural norm that an owner/maintainer should publicly hand title to someone if he or she can no longer maintain the project.

The simplest non-trivial case is when a project has multiple co-maintainers working under a single `benevolent dictator' who owns the project. Custom favors this mode for group projects; it has been shown to work on projects as large as the Linux kernel or Emacs, and solves the ``who decides'' problem in a way that is not obviously worse than any of the alternatives.

Typically, a benevolent-dictator organization evolves from an owner-maintainer organization as the founder attracts contributors. Even if the owner stays dictator, it introduces a new level of possible disputes over who gets credited for what parts of the project.

In this situation, custom places an obligation on the owner/dictator to credit contributors fairly (through, for example, appropriate mentions in README or history files). In terms of the Lockean property model, this means that by contributing to a project you earn part of its reputation return (positive or negative).

Pursuing this logic, we see that a `benevolent dictator' does not in fact own his entire project unqualifiedly. Though he has the right to make binding decisions, he in effect trades away shares of the total reputation return in exchange for others' work. The analogy with sharecropping on a farm is almost irresistible, except that a contributor's name stays in the credits and continues to `earn' to some degree even after that contributor is no longer active.

As benevolent-dictator projects add more participants, they tend to develop two tiers of contributors; ordinary contributors and co-developers. A typical path to becoming a co-developer is taking responsibility for a major subsystem of the project. Another is to take the role of `lord high fixer', characterizing and fixing many bugs. In this way or others, co-developers are the contributors who make a substantial and continuing investment of time in the project.

The subsystem-owner role is particularly important for our analysis and deserves further examination. Hackers like to say that `authority follows responsibility'. A co-developer who accepts maintainance responsibility for a given subsystem generally gets to control both the implementation of that subsystem and its interfaces with the rest of the project, subject only to correction by the project leader (acting as architect). We observe that this rule effectively creates enclosed properties on the Lockean model within a project, and has exactly the same conflict-prevention role as other property boundaries.

By custom, the `dictator' or project leader in a project with co-developers is expected to consult with those co-developers on key decisions. This is especially so if the decision concerns a subsystem which a co-developer `owns' (that is, has invested time in and taken responsibility for). A wise leader, recognizing the function of the project's internal property boundaries, will not lightly interfere with or reverse decisions made by subsystem owners.

Some very large projects discard the `benevolent dictator' model entirely. One way to do this is turn the co-developers into a voting committee (as with Apache). Another is rotating dictatorship, in which control is occasionally passed from one member to another within a circle of senior co-developers; the Perl developers organize themselves this way.

Such complicated arrangements are widely considered unstable and difficult. Clearly this perceived difficulty is largely a function of the known hazards of design-by-committee, and of committees themselves; these are problems the hacker culture consciously understands. However, I think some of the visceral discomfort hackers feel about committee or rotating-chair organizations is because they're hard to fit into the unconscious Lockean model hackers use for reasoning about the simpler cases. It's problematic, in these complex organizations, to do an accounting of either ownership in the sense of control or ownership of reputation returns. It's hard to see where the internal boundaries are, and thus hard to avoid conflict unless the group enjoys an exceptionally high level of harmony and trust.

17. Conflict and Conflict Resolution

We've seen that within projects, an increasing complexity of roles is expressed by a distribution of design authority and partial property rights. While this is an efficient way to distribute incentives, it also dilutes the authority of the project leader --

most importantly, it dilutes the leader's authority to squash potential conflicts.

While technical arguments over design might seem the most obvious risk for internecine conflict, they are seldom a serious cause of strife. These are usually relatively easily resolved by the territorial rule that authority follows responsibility.

Another way of resolving conflicts is by seniority -- 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 (that is, the side with the most property rights in the whole project) wins.

(Equivalently, the side with the least invested loses. Interestingly this happens to be the same heuristic that many relational database engines resolve deadlocks. When two threads are deadlocked over resources, the side with the least invested in the current transaction is selected as the deadlock victim and is terminated. This usually selects the longest running transaction, or the more senior, as the victor.)

These rules generally suffice to resolve most project disputes. When they do not, fiat of the project leader usually suffices. Disputes that survive both these filters are rare.

Conflicts do not as a rule become serious unless these two criteria ("authority follows responsibility" and "seniority wins") point in different directions, *and* the authority of the project leader is weak or absent. The most obvious case in which this may occur is a succession dispute following the disappearance of the project lead. I have been in one fight of this kind. It was ugly, painful, protracted, only resolved when all parties became exhausted enough to hand control to an outside person, and I devoutly hope I am never anywhere near anything of the kind again.

Ultimately, all of these conflict-resolution mechanisms rest on the wider hacker community's willingness to enforce them. The only available enforcement mechanisms are flaming and shunning – public condemnation of those who break custom, and refusal to cooperate with them after they have done so.

18. Acculturation Mechanisms and the Link to Academia

An early version of this paper posed the following research question: How does the community inform and instruct its members as to its customs? Are the customs self-evident or self-organizing at a semi-conscious level, are they taught by example, are they taught by explicit instruction?

Teaching by explicit instruction is clearly rare, if only because few explicit descriptions of the culture's norms have existed to be used up to now.

Many norms are taught by example. To cite one very simple case, there is a norm that every software distribution should have a file called README or READ.ME that contains first-look instructions for browsing the distribution. This convention has been well established since at least the early 1980s; it has even, occasionally, been written down. But one normally derives it from looking at many distributions.

On the other hand, some hacker customs are self-organizing once one has acquired a basic (perhaps unconscious) understanding of the reputation game. Most hackers never have to be taught the three taboos I listed earlier in this paper, or at least would claim if asked that they are self-evident rather than transmitted. This phenomenon invites closer analysis -- and perhaps we can find its explanation in the process by which hackers acquire knowledge about the culture.

Many cultures use hidden clues (more precisely `mysteries' in the religio/mystical sense) as an acculturation mechanism. These are secrets which are not revealed to outsiders, but are expected to be discovered or deduced by the aspiring newbie. To be accepted inside, one must demonstrate that one both understands the mystery and has learned it in a culturally approved way.

The hacker culture makes unusually conscious and extensive use of such clues or tests. We can see this process operating at at least three levels:

Password-like specific mysteries. As one example, there is a USENET newsgroup called alt.sysadmin.recovery that has a very explicit such secret; you cannot post without knowing it, and knowing it is considered evidence you are fit to post. The regulars have a strong taboo against revealing this secret.

The requirement of initiation into certain technical mysteries. One must absorb a good deal of technical knowledge before one can give valued gifts (e.g. one must know at least one of the major computer languages). This requirement functions in the large in the way hidden clues do in the small, as a filter for qualities (such as capability for abstract thinking, persistence, and mental flexibility) which are necessary to function in the culture.

Social-context mysteries. One becomes involved in the culture through attaching oneself to specific projects. Each project is a live social context of hackers which the would-be contributor has to investigate and understand socially as well as technically in order to function. (Concretely, a common way one does this is by reading the project's Web pages and/or email archives.) It is through these project groups that newbies experience the behavioral example of experienced hackers.

In the process of acquiring these mysteries, the would-be hacker picks up contextual knowledge which (after a while) does make the three taboos and other customs seem `self-evident'.

One might, incidentally, argue that the structure of the hacker gift culture itself is its own central mystery. One is not considered acculturated (concretely: no one will call you a hacker) until one demonstrates a gut-level understanding of the reputation game and its implied customs, taboos, and usages. But this is trivial; all cultures demand such understanding from would-be joiners. Furthermore the hacker culture evinces no desire to have its internal logic and folkways kept secret -- or, at least, nobody has ever flamed me for revealing them!

Respondents to this paper too numerous to list have pointed out that hacker ownership customs seem intimately related to (and may derive directly from) the practices of the academic world, especially the scientific research commmunity. This research community has similar problems in mining a territory of potentially productive ideas, and exhibits very similar adaptive solutions to those problems in the ways it uses peer review and reputation.

Since many hackers have had formative exposure to academia (it's common to learn how to hack while in college) the extent to which academia shares adaptive patterns with the hacker culture is of more than casual interest in understanding how these customs are applied.

Obvious parallels with the hacker `gift culture' as I have characterized it abound in academia. Once a researcher achieves tenure, there is no need to worry about survival issues.

(Indeed, the concept of tenure can probably be traced back to an earlier gift culture in which ``natural philosophers'' were primarily wealthy gentlemen with time on their hands to devote to research.) In the absence of survival issues, reputation enhancement becomes the driving goal, which encourages sharing of new ideas and research through journals and other media. This makes objective functional sense because scientific research, like the hacker culture, relies heavily on the idea of `standing upon the shoulders of giants', and not having to rediscover basic principles over and over again.

Some have gone so far as to suggest that hacker customs are merely a reflection of the research community's folkways and have actually (in most cases) been acquired there by individual hackers. This probably overstates the case, if only because hacker custom seems to be readily acquired by intelligent high-schoolers!

19. Gift Outcompetes Exchange

There is a more interesting possibility here. I suspect academia and the hacker culture share adaptive patterns not because they're genetically related, but because they've both evolved the one most optimal social organization for what they're trying to do, given the laws of nature and and the instinctive wiring of human beings. The verdict of history seems to be that free-market capitalism is the globally optimal way to cooperate for economic efficiency; perhaps, in a similar way, the reputation-game gift culture is the globally optimal way to cooperate for generating (and checking!) high-quality creative work.

Support for this theory becomes from a large body of psychological studies on the interaction between art and reward [GNU]. These studies have received less attention than they should, in part perhaps because their popularizers have shown a tendency to overinterpret them into general attacks against the free market and intellectual property. Nevertheless, their results do suggest that some kinds of scarcity-economics rewards actually decrease the productivity of creative workers such as programmers.

Psychologist Theresa Amabile of Brandeis University, cautiously summarizing the results of a 1984 study of motivation and reward, observed ``It may be that commissioned work will, in general, be less creative than work that is done out of pure interest.''. Amabile goes on to observe that ``The more complex the activity, the more it's hurt by extrinsic reward.'' Interestingly, the studies suggest that flat salaries don't demotivate, but piecework rates and bonuses do.

Thus, it may be economically smart to give performance bonuses to people who flip burgers or dug ditches, but it's probably smarter to decouple salary from performance in a programming shop and let peeople choose their own projects (both trends that the open-source world takes to their logical conclusions). Indeed, these results suggest that the only time it is a good idea to reward performance in programming is when the programmer is so motivated that he or she would have worked without the reward!

Other researchers in the field are willing to point a finger straight at the issues of autonomy and creative control that so preoccupy hackers. ``To the extent one's experience of being self-determined is limited,'' said Richard Ryan, associate psychology professor at the University of Rochester, ``one's creativity will be reduced as well.''

In general, presenting any task as a means rather than an end in itself seems to demotivate. Even winning a competition with others or gaining peer esteem can be demotivating in this way if it is experienced as work for reward (which may explain why hackers are culturally prohibited from explicitly seeking or claiming that esteem).

To complicate the management problem further, controlling verbal feedback seems to be just as demotivating as piecework payment. Ryan found that corporate employees who were told, ``Good, you're doing as you *should*'' were ``significantly less intrinsically motivated than those who received feedback informationally.''

It may still be intelligent to offer incentives, but they have to come without attachments to avoid gumming up the works. There is a criticl difference (Ryan observes) between saying, ``I'm giving you this reward because I recognize the value of your work'' and ``You're getting this reward because you've lived up to my standards.'' The first does not demotivate; the second does.

In these psychological observations we can ground a case that an open-source development group will be substantially more productive (especially over the long term, in which creativity becomes more critical as a productivity multiplier) than an equivalently sized and skilled group of closed-source programmers (de)motivated by scarcity rewards.

This suggests from a slightly different angle one of the speculations in *The Cathedral And The Bazaar*; that, ultimately, the industrial/factory mode of software production was doomed to be outcompeted from the moment capitalism began to create enough of a wealth surplus that many programmers could live in a post-scarcity gift culture.

Indeed, it seems the prescription for highest software productivity is almost a Zen paradox; if you want the most efficient production, you must give up trying to *make* programmers produce. Handle their subsistence, give them their heads, and forget about deadlines. To a conventional manager this sounds crazily indulgent and doomed -- but it is *exactly* the recipe with which the open-source culture is now clobbering its competition.

20. Conclusion: From Custom to Customary Law

We have examined the customs which regulate the ownership and control of open-source software. We have seen how they imply an underlying theory of property rights homologous to the Lockean theory of land tenure. We have related that to an analysis of the hacker culture as a `gift culture' in which participants compete for prestige by giving time, energy, and creativity away. We have examined the implications of this analysis for conflict resolution in the culture.

The next logical question to ask is "Why does this matter?" Hackers developed these customs without conscious analysis and (up to now) have followed them without conscious analysis. It's not immediately clear that conscious analysis has gained us anything practical -- unless, perhaps, we can move from description to prescription and deduce ways to improve the functioning of these customs.

We have found a close logical analogy for hacker customs in the theory of land tenure under the Anglo-American common-law tradition. Historically [Miller], the European tribal cultures that invented this tradition improved their dispute-resolution systems by moving from a system of unarticulated, semi-conscious custom to a body of explicit customary law memorized by tribal wisemen – and eventually, written down.

Perhaps, as our population rises and acculturation of all new members becomes more difficult, it is time for the hacker culture to do something analogous -- to develop written codes of good practice for resolving the various sorts of disputes that can arise in connection with open-source projects, and a tradition of arbitration in which senior members of the community may be asked to mediate disputes.

The analysis in this paper suggests the outlines of what such a code might look like, making explicit that which was previously implicit. No such codes could be imposed from above; they would have to be voluntarily adopted by the founders or owners of individual projects. Nor could they be completely rigid, as the pressures on the culture are likely to change over time. Finally, for enforcement of such codes to work, they would have to reflect a broad consensus of the hacker tribe.

I have begun work on such a code, tentatively titled the "Malvern Protocol" after the little town where I live. If the general analysis in this paper becomes sufficiently widely accepted, I will make the Malvern Protocol publicly available as a model code for dispute resolution. Parties interested in critiquing and developing this code, or just offering feedback on whether they think it's a good idea or not, are invited to contact me by email.

21. Questions for Further Research

The culture's (and my own) understanding of large projects that don't follow a benevolent-dictator model is weak. Most such projects fail. A few become spectacularly successful and important (Perl, Apache, KDE). Nobody really understands where the difference lies. There's a vague sense abroad that each such project is *sui generis* and stands or falls on the group dynamic of its particular members, but is this true or are there replicable strategies a group can follow?

22. Bibliography

[Miller] Miller, William Ian; *Bloodtaking and Peacemaking: Feud, Law, and Society in Saga Iceland*; University of Chicago Press 1990, ISBN 0-226-52680-1. A fascinating study of Icelandic folkmoot law, which both illuminates the ancestry of the Lockean theory of property and describes the later stages of a historical process by which custom passed into customary law and thence to written law.

[Mal] Malaclypse the Younger; *Principia Discordia, or How I Found Goddess and What I Did To Her When I Found Her*; Loompanics, ISBN 1-55950-040-9. There is much enlightening silliness to be found in Discordianism. Amidst it, the `SNAFU principle' provides a rather trenchant analysis of why command hierarchies don't scale well. There's a browseable HTML version.

[BCT] J. Barkow, L. Cosmides, and J. Tooby (Eds.); *The adapted mind: Evolutionary psychology and the generation of culture.* New York: Oxford University Press 1992. An excellent introduction to evolutionary psychology. Some of the papers bear directly on the three cultural types I discuss (command/exchange/gift), suggesting that these patterns are wired into the human psyche fairly deep.

[MHG] Goldhaber, Michael K.; The Attention Economy and the Net. I discovered this paper after my version 1.7. It has obvious flaws (Goldhaber's argument for the inapplicability of economic reasoning to attention does not bear close examination), but Goldhaber nevertheless has funny and perceptive things to say about the role of attention-seeking in organizing behavior. The prestige or peer repute I have discussed can fruitfully be viewed as a particular case of attention in his sense.

23. Endnotes

[N] The term `noosphere' is an obscure term of art in philosophy It is pronounced KNOW-uh-sfeer (two o-sounds, one long and stressed, one short and unstressed tending towards schwa). If one is being excruciatingly correct about one's orthography, it is properly spelled with a diaresis over the second `o' to mark it as a separate vowel.

In more detail; this term for ``the sphere of human thought'' derives from the Greek `nous' meaning `mind', `spirit', or `breath'. It was invented by E. LeRoy in *Les origines humaines et l'evolution de l'intelligence* (Paris 1928). It was popularized first by the Russian biologist and pioneering ecologist Vladimir Ivanovich Vernadsky, (1863-1945), then by the Jesuit paleontologist/philosopher Pierre Teilhard de Chardin (1881-1955). It is with de Chardin's theory of future human evolution to a form of pure mind culminating in union with the Godhead that the term is now primarily associated.

[DF] David Friedman, one of the most lucid and accessible thinkers in contemporary economics, has written an excellent outline of the history and logic of intellectual-property law. I recommend it as a starting point to anyone interested in these issues.

[SP] One interesting difference between the Linux and BSD worlds is that the Linux kernel (and associated OS core utilities) have never forked, but BSD's has, at least three times. What makes this interesting is that the social structure of the BSD groups is centralized in a way intended to define clear lines of authority and to prevent forking, while the decentralized and amorphous Linux community takes no such measures. It appears that the projects which open up development the most actually have the *least* tendency to fork!

Henry Spencer <henry@spsystems.net> suggests that, in general, the stability of a political system is inversely proportional to the height of the entry barriers to its political process. His analysis is worth quoting here:

One major strength of a relatively open democracy is that most potential revolutionaries find it easier to make progress toward their objectives by working via the system rather by attacking it. This strength is easily undermined if established parties act together to `raise the bar', making it more difficult for small dissatisfied groups to see *some* progress made toward their goals.

(A similar principle can be found in economics. Open markets have the strongest competition, and generally the best and cheapest products. Because of this, it's very much in the best interests of established companies to make market entry more difficult -- for example, by convincing governments to require elaborate RFI testing on computers, or by creating `consensus' standards which are so complex that they cannot be implemented effectively from scratch without large resources. The markets with the strongest entry barriers are the ones that come under the strongest attack from revolutionaries, e.g. the Internet and the Justice Dept. vs. the Bell System.)

An open process with low entry barriers encourages participation rather than secession, because one can get results without the high overheads of secession. The results may not be as impressive as what could be achieved by seceding, but they come at a lower price, and most people will consider that an acceptable tradeoff. (When the Spanish government revoked Franco's anti-Basque laws and offered the Basque provinces their own schools and limited local autonomy, most of the

Basque Separatist movement evaporated almost overnight. Only the hard-core Marxists insisted that it wasn't good enough.)

[RP] There are some subtleties about rogue patches. One can divide them into `friendly' and `unfriendly' types. A `friendly' patch is designed to be merged back into the project's main-line sources under the maintainer's control (whether or not that merge actually happens); an `unfriendly' one is intended to yank the project in a direction the maintainer doesn't approve. Some projects (notably the Linux kernel itself) are pretty relaxed about friendly patches and even encourage independent distribution of them as part of their beta-test phase. An unfriendly patch, on the other hand, represents a decision to compete with the original and is a serious matter. Maintaining a whole raft of unfriendly patches tends to lead to forking.

[LW] I am indebted to Michael Funk <mwfunk@uncc.campus.mci.net> for pointing out how instructive a contrast with hackers the pirate culture is. Linus Walleij has posted an analysis of their cultural dynamics that differs from mine (describing them as a scarcity culture) in A Comment on `Warez D00dz' Culture

The contrast may not last. Former cracker Andrej Brandt <andy@pilgrim.cs.net.pl> reports that he believes the cracker/warez d00dz culture is now withering away, with its brightest people and leaders assimilating to the open-source world. Independent evidence for this view may be provided by a precedent-breaking July 1999 action of the cracker group calling itself `Cult of the Dead Cow'. They have released their `Back Orifice 2000' for breaking Microsoft Windows security tools under the GPL.

[HT] In evolutionary terms, the craftsman's urge itself may (like internalized ethics) be a result of the high risk and cost of deception. Evolutionary psychologists have collected experimental evidence [BCT] that human beings have brain logic specialized for detecting social deceptions, and it is fairly easy to see why our ancestors should have been selected for ability to detect cheating. Therefore, if one wishes to have a reputation for personality traits which confer advantage but are risky or costly, it may actually be better tactics to actually have these traits than to fake them. (``Honesty is the best policy'')

Evolutionary psychologists have suggested that this explains behavior like barroom fights. Among younger adult male humans, having a reputation for `toughness' is both socially and (even in today's feminist-influenced climate) sexually useful. Faking "toughness", however, is extremely risky; the negative result of being found out leaves one in a worse position than never having claimed the trait. The cost of deception is so high that it is sometimes better minimaxing to internalize `toughness' and risk serious injury in a fight to prove it. Parallel observations have been made about less controversial traits like `honesty'.

Though the primary meditation-like rewards of creative work should not be underestimated, the craftsman's urge is probably at least in part just such an internalization (where the base trait is `capacity for painstaking work' or something similar).

[MH] A concise summary of Maslow's hierarchy and related theories is available on the Web at Maslow's Hierarchy of Needs

[DC] However, demanding humility from leaders may be a more general characteristic of gift or abundance cultures. David Christie <dc@netscape.com> reports on a trip through the outer islands of Fiji: ``In Fijian village chiefs, we observed the same sort of self-deprecating, low-key leadership style that you attribute to open source project leaders. [...] Though accorded great respect and of course all of whatever actual power there is in Fiji, the chiefs we met demonstrated genuine humility and often a saint-like acceptance of their duty. This is particularly interesting given that being chief is a hereditary role, not an elected position or a popularity contest. Somehow they are trained to it by the culture itself, although they are born to it, not chosen by their peers.'' He goes on to emphasize that he believes the characteristic style of Fijian chiefs springs from the difficulty of compelling cooperation: a chief has ``no big carrot or big stick''.

[NO] As a matter of observable fact, people who found successful projects gather more prestige than people who do arguably equal amounts of work debugging and assisting with successful projects. An earlier version of this paprer asked ``Is this a rational valuation of comparative effort, or is it a second-order effect of the unconscious territorial model we have adduced here?'' Several respondents suggested persuasive and essentially equivalent theories. The following analysis by Ryan Waldron <rew@erebor.com> puts the case well:

In the context of the Lockean land theory, one who establishes a new and successful project has essentially discovered or opened up new territory on which others can homestead. For most successful projects, there is a pattern of declining returns, so that after a while, the credit for contributions to a project has become so diffuse that it is hard for significant reputation to accrete to a late participant, regardless of the quality of his work.

For instance, how good a job would I have to do making modifications to the perl code to have even a fraction of the recognition for my participation that Larry, Tom, Randall, and others have achieved?

However, if a new project is founded [by someone else] tomorrow, and I am an early and frequent participant in it, my ability to share in the respect generated by such a successful project is greatly enhanced by my early participation therein (assuming similar quality of contributions). I reckon it to be similar to those who invest in Microsoft stock early and those who invest in it later. Everyone may profit, but early participants profit more. Therefore, at some point I will be more interested in a new and successful IPO than I will be in participating in the continual increase of an existing body of corporate stock.

Ryan Waldron's analogy can be extended. The project founder has to a missionary sell of a new idea which may or may not be acceptable or of use to others. Thus the founder incurs something analogous to an IPO risk (of possible damage to their reputation), more so than others who assist with a project that has already garnered some acceptance by their peers. The founder's reward is consistent despite the fact that the assistants may be putting in more work in real terms. This is easily seen as analogous to the relationship between risk and rewards in an exchange economy.

Other respondents have observed that our nervous system is tuned to perceive differences, not steady state. The revolutionary change evidenced by the creation of a new project is therefore much more noticeable than the cumulative effect of constant incremental improvement. Thus Linus is revered as the father of Linux, although the net effect of improvements by thousands of other contributors have done more to contribute to the success of the OS than one man's work ever could.

[HD] The phrase ``de-commoditizing'' is a reference to the Halloween Documents in which Microsoft used ``de-commoditize'' quite frankly to refer to their most effective long-term strategy for maintaining an exploitative monopoly lock on customers.

[GNU] The Free Software Foundation's main web site carries an article that summarizes the results of many of these studies. The quotes in this paper are excerpted from there.

24. Acknowledgements

Robert Lanphier <robla@real.com> contributed much to the discussion of egoless behavior. Eric Kidd <eric.kidd@pobox.com> highlighted the role of valuing humility in preventing cults of personality. The section on global effects was inspired by comments from Daniel Burn <daniel@tsathoggua.lab.usyd.edu.au>. Mike Whitaker <mrw@entropic.co.uk> inspired the main thread in the section on acculturation. Chris Phoenix <cphoenix@best.com> pointed out the importance of the fact that hackers cannot gain reputation by doing other hackers down.

I am solely responsible for what has gone into this paper, and any errors or misconceptions. However, I have welcomed comments and feedback and used them to improve the paper -- a process which I do not expect to end at any predefined time.

The Magic Cauldron

1. Indistinguishable From Magic

In Welsh myth, the goddess Ceridwen owned a great cauldron which would magically produce nourishing food -- when commanded by a spell known only to the goddess. In modern science, Buckminster Fuller gave us the concept of `ephemeralization', technology becoming both more effective and less expensive as the physical resources invested in early designs are replaced by more and more information content. Arthur C. Clarke connected the two by observing that ``Any sufficiently advanced technology is indistinguishable from magic".

To many people, the successes of the open-source community seem like an implausible form of magic. High-quality software materializes ``for free", which is nice while it lasts but hardly seems sustainable in the real world of competition and scarce resources. What's the catch? Is Ceridwen's cauldron just a conjuring trick? And if not, how does ephemeralization work in this context – what spell is the goddess speaking?

2. Beyond Geeks Bearing Gifts

The experience of the open-source culture has certainly confounded many of the assumptions of people who learned about software development outside it. ``The Cathedral and the Bazaar" [CatB] described the ways in which decentralized cooperative software development effectively overturns Brooks's Law, leading to unprecedented levels of reliability and quality on individual projects. ``Homesteading the Noosphere" [HtN] examined the social dynamics within which this `bazaar' style of development is situated, arguing that it is most effectively understood not in conventional exchange-economy terms but as what anthropologists call a `gift culture' in which members compete for status by giving things away. In this paper we shall begin by exploding some common myths about software production economics; then continue the analysis of [CatB] and [HtN] into the realm of economics, game theory and business models, developing new conceptual tools needed to understand the way that the gift culture of open-source developers can sustain itself in an exchange economy.

In order to pursue this line of analysis without distraction, we'll need to abandon (or at least agree to temporarily ignore) the `gift culture' level of explanation. [HtN] posited that gift culture behavior arises in situations where survival goods are abundant enough to make the exchange game no longer very interesting; but while this appears sufficiently powerful as a **psychological** explanation of behavior, it lacks suffiency as an explanation of the mixed **economic** context in which most open-source developers actually operate. For most, the exchange game has lost its appeal but not its power to constrain. Their behavior has to make sufficient material-scarcity-economics sense to keep them in a gift-culture-supporting zone of surplus.

Therefore, we now will consider (from entirely **within** the realm of scarcity economics) the modes of cooperation and exchange that sustain open-source development. While doing so we will answer the pragmatic question ``How do I make money at this?", in detail and with examples. First, though, we will show that much of the tension behind that question derives from prevailing folk models of software-production economics that are false to fact.

(A final note before the exposition: the discussion and advocacy of open-source development in this paper should not be construed as a case that closed-source development is intrinsically wrong, nor as a brief against intellectual-property rights in software, nor as an altruistic appeal to `share'. While these arguments are still beloved of a vocal minority in the open-source development community, experience since [CatB] has made it clear that they are unnecessary. An entirely sufficient case for open-source development rests on its engineering and economic outcomes -- better quality, higher reliability, lower costs, and increased choice.)

3. The Manufacturing Delusion

We need to begin by noticing that computer programs like all other kinds of tools or capital goods, have two distinct kinds of economic value. They have **use value** and **sale value**.

The **use value** of a program is its economic value as a tool. The **sale value** of a program is its value as a salable commodity. (In professional economist-speak, sale value is value as a final good, and use value is value as an intermediate good.)

When most people try to reason about software-production economics, they tend to assume a `factory model' that is founded on the following fundamental premises.

1. Most developer time is paid for by sale value.

2. The sale value of software is proportional to its development cost (i.e. the cost of the resources required to functionally replicate it) and to its use value.

In other words, people have a strong tendency to assume that software has the value characteristics of a typical manufactured good. But both of these assumptions are demonstrably false.

First, code written for sale is only the tip of the programming iceberg. In the pre-microcomputer era it used to be a commonplace that 90% of all the code in the world was written in-house at banks and insurance companies. This is probably no longer the case – other industries are much more software-intensive now, and the finance industry's share of the total has accordingly dropped -- but we'll see shortly that there is empirical evidence that around 95% of code is still written in-house.

This code includes most of the stuff of MIS, the financial- and database-software customizations every medium and large company needs. It includes technical-specialist code like device drivers (almost nobody makes money selling device drivers, a point we'll return to later on). It includes all kinds of embedded code for our increasingly microchip-driven machines - from machine tools and jet airliners to cars to microwave ovens and toasters.

Most such in-house code is integrated with its environment in ways that make reusing or copying it very difficult. (This is true whether the `environment' is a business office's set of procedures or the fuel-injection system of a combine harvester.) Thus, as the environment changes, there is a lot of work continually needed to keep the software in step.

This is called `maintenance', and any software engineer or systems analyst will tell you that it makes up the vast majority (more than 75%) of what programmers get paid to do. Accordingly, most programmer-hours are spent (and most programmer salaries are paid for) writing or maintaining in-house code that has no sale value at all – a fact the reader may readily check by examining the listings of programming jobs in any newspaper with a `Help Wanted' section.

Scanning the employment section of your local newspaper is an enlightening experiment which I urge the reader to perform for him- or herself. Examine the jobs listings under programming, data processing, and software engineering for positions that involve the development of software. Categorize each such job according to whether the software is being developed for use or for sale.

It will quickly become clear that, even given the most inclusive definition of `for sale', at least nineteen in twenty of the salaries offered are being funded strictly by use value (that is, value as an intermediate good). This is our reason for believing that only 5% of the industry is sale-value-driven. Note, however, that the rest of the analysis in this paper is relatively insensitive to this number; if it were 15% or even 20%, the economic consequences would remain essentially the same.

(When I speak at technical conferences, I usually begin my talk by asking two questions: how many in the audience are paid to write software, and for how many do their salaries depend on the sale value of software. I generally get a forest of hands for the first question, few or none for the second, and considerable audience surprise at the proportion.)

Second, the theory that the sale value of software is coupled to its development or replacement costs is even more easily demolished by examining the actual behavior of consumers. There are many goods for which a proportion of this kind actually holds (before depreciation) -- food, cars, machine tools. There are even many intangible goods for which sale value couples strongly to development and replacement cost -- rights to reproduce music or maps or databases, for example. Such goods may retain or even increase their sale value after their original vendor is gone.

By contrast, when a software product's vendor goes out of business (or if the product is merely discontinued), the maximum price consumers will pay for it rapidly goes to near zero regardless of its theoretical use value or the development cost of a functional equivalent. (To check this assertion, examine the remainder bins at any software store near you.)

The behavior of retailers when a vendor folds is very revealing. It tells us that they know something the vendors don't. What they know is this: the price a consumer will pay is effectively capped by the **expected future value of vendor service** (where `service' is here construed broadly to include enhancements, upgrades, and follow-on projects).

In other words, software is largely a service industry operating under the persistent but unfounded delusion that it is a manufacturing industry.

It is worth examining why we normally tend to believe otherwise. It may simply be because the small portion of the software industry that manufactures for sale is also the only part that advertises its product. Also, some of the most visible and heavily advertised products are ephemera like games that have little in the way of continuing service requirements (the exception, rather than the rule) [SH].

It is also worth noting that the manufacturing delusion encourages price structures that are pathologically out of line with the actual breakdown of development costs. If (as is generally accepted) over 75% of a typical software project's life-cycle costs will be in maintenance and debugging and extensions, then the common price policy of charging a high fixed purchase price and relatively low or zero support fees is bound to lead to results that serve all parties poorly.

Consumers lose because, even though software is a service industry, the incentives in the factory model all cut against a vendor's offering **competent** service. If the vendor's money comes from selling bits, most effort will go to making bits and shoving them out the door; the help desk, not a profit center, will become a dumping ground for the least effective and get only enough resources to avoid actively alienating a critical number of customers.

The other side of this coin is that most vendors buying this factory model will also fail in the longer run. Funding indefinitely-continuing support expenses from a fixed price is only viable in a market that is expanding fast enough to cover the support and life-cycle costs entailed in yesterday's sales with tomorrow's revenues. Once a market matures and sales slow down, most vendors will have no choice but to cut expenses by orphaning the product.

Whether this is done explicitly (by discontinuing the product) or implicitly (by making support hard to get), it has the effect of driving customers to competitors (because it destroys the product's expected future value, which is contingent on that service). In the short run, one can escape this trap by making bug-fix releases pose as new products with a new price attached, but consumers quickly tire of this. In the long run, therefore, the only way to escape is to have no competitors -- that is, to have an effective monopoly on one's market. In the end, there can be only one.

And, indeed, we have repeatedly seen this support-starvation failure mode kill off even strong second-place competitors in a market niche. (The pattern should be particularly clear to anyone who has ever surveyed the history of proprietary PC operating systems, word processors, accounting programs or business software in general.) The perverse incentives set up by the factory model lead to a winner-take-all market dynamic in which even the winner's customers end up losing.

If not the factory model, then what? To handle the real cost structure of the software life-cycle efficiently (in both the informal and economics-jargon senses of `efficiency'), we require a price structure founded on service contracts, subscriptions, and a **continuing** exchange of value between vendor and customer. Under the efficiency-seeking conditions of the free market, therefore, we can predict that this is the sort of price structure most of a mature software industry will ultimately follow.

The foregoing begins to give us some insight into why open-source software increasingly poses not merely a technological but an economic challenge to the prevailing order. The effect of making software `free', it seems, is to force us into that service-fee-dominated world -- and to expose what a relatively weak prop the sale value of closed-source bits was all along.

The term `free' is misleading in another way as well. Lowering the cost of a good tends to increase, rather than decrease, total investment in the infrastructure that sustains it. When the price of cars goes down, the demand for auto mechanics goes up -- which is why even those 5% of programmers now compensated by sale-value would be unlikely to suffer in an open-source world. The people who lose in the transition won't be programmers, they will be investors who have bet on closed-source strategies where they're not appropriate.

4. The ``information wants to be free" Myth

There is another myth, equal and opposite to the factory-model delusion, which often confuses peoples' thinking about the economics of open-source software. It is that ``information wants

to be free''. This usually unpacks to a claim that the zero marginal cost of reproducing digital information implies that its clearing price ought to be zero.

The most general form of this myth is readily exploded by considering the value of information that constitutes a claim on a rivalrous good -- a treasure map, say, or a Swiss bank account number, or a claim on services such as a computer account password. Even though the claiming information can be duplicated at zero cost, the item being claimed cannot be. Hence, the non-zero marginal cost for the item can be inherited by the claiming information.

We mention this myth mainly to assert that it is unrelated to the economic-utility arguments for open source; as we'll see later, those would generally hold up well even under the assumption that software actually **does** have the (nonzero) value structure of a manufactured good. We therefore have no need to tackle the question of whether software `should' be free or not.

5. The Inverse Commons

Having cast a skeptical eye on one prevailing model, let's see if we can build another -- a hard-nosed economic explanation of what makes open-source cooperation sustainable.

This is a question that bears examination on a couple of different levels. On one level, we need to explain the behavior of individuals who contribute to open-source projects; on another, we need to understand the economic forces that sustain cooperation on open-source projects like Linux or Apache.

Again, we must first demolish a widespread folk model that interferes with understanding. Over every attempt to explain cooperative behavior there looms the shadow of Garret Hardin's Tragedy of the Commons.

Hardin famously asks us to imagine a green held in common by a village of peasants, who graze their cattle there. But grazing degrades the commons, tearing up grass and leaving muddy patches, which re-grow their cover only slowly. If there is no agreed-on (and enforced!) policy to allocate grazing rights that prevents overgrazing, all parties' incentives push them to run as many cattle as quickly as possible, trying to extract maximum value before the commons degrades into a sea of mud.

Most people have an intuitive model of cooperative behavior that goes much like this. It's not actually a good diagnosis of the economic problems of open-source, which are free-rider (underprovision) rather than congested-public-good (overuse). Nevertheless, it is the analogy I hear behind most off-the-cuff objections.

The tragedy of the commons predicts only three possible outcomes. One is the sea of mud. Another is for some actor with coercive power to enforce an allocation policy on behalf of the village (the communist solution). The third is for the commons to break up as village members fence off bits they can defend and manage sustainably (the property-rights solution).

When people reflexively apply this model to open-source cooperation, they expect it to be unstable with a short half-life. Since there's no obvious way to enforce an allocation policy for programmer time over the internet, this model leads straight to a prediction that the commons will break up, with various bits of software being taken closed-source and a rapidly decreasing amount of work being fed back into the communal pool.

In fact, it is empirically clear that the trend is opposite to this. The breadth and volume of open-source development (as measured by, for example, submissions per day at Metalab or announcements per day at freshmeat.net) is steadily increasing. Clearly there is some critical way in which the ``Tragedy of the Commons'' model fails to capture what is actually going on.

Part of the answer certainly lies in the fact that using software does not decrease its value. Indeed, widespread use of open-source software tends to **increase** its value, as users fold in their own fixes and features (code patches). In this inverse commons, the grass grows taller when it's grazed on.

Another part of the answer lies in the fact that the putative market value of small patches to a common source base is hard to capture. Supposing I write a fix for an irritating bug, and suppose many people realize the fix has money value; how do I collect from all those people? Conventional payment systems have high enough overheads to make this a real problem for the sorts of micropayments that would usually be appropriate.

It may be more to the point that this value is not merely hard to capture, in the general case it's hard to even **assign**. As a thought experiment let us suppose that the Internet came equipped with the theoretically ideal micropayment system -- secure, universally accessible, zero-overhead. Now let's say you have written a patch labeled ``Miscellaneous Fixes to the Linux Kernel''. How do you know what price to ask? How would a potential buyer, not having seen the patch yet, know what is reasonable to pay for it?

What we have here is almost like a funhouse-mirror image of F.A. Hayek's `calculation problem' -- it would take a superbeing, both able to evaluate the functional worth of patches and trusted to set prices accordingly, to lubricate trade.

Unfortunately, there's a serious superbeing shortage, so patch author J. Random Hacker is left with two choices: sit on the patch, or throw it into the pool for free. The first choice gains nothing. The second choice may gain nothing, or it may encourage reciprocal giving from others that will address some of J. Random's problems in the future. The second choice, apparently altruistic, is actually optimally selfish in a game-theoretic sense.

In analyzing this kind of cooperation, it is important to note that while there is a free-rider problem (work may be underprovided in the absence of money or money-equivalent compensation) it is not one that scales with the number of end-users. The complexity and communications overhead of an open-source project is almost entirely a function of the number of developers involved; having more end-users who never look at source costs effectively nothing. It may increase the rate of silly questions appearing on the project mailing lists, but this is relatively easily forestalled by maintaining a Frequently Asked Questions list and blithely ignoring questioners who have obviously not read it (and in fact both these practices are typical).

The real free-rider problems in open-source software are more a function of friction costs in submitting patches than anything else. A potential contributor with little stake in the cultural reputation game (see [HtN]) may, in the absence of money compensation, think ``It's not worth submitting this fix because I'll have to clean up the patch, write a ChangeLog entry, and sign the FSF assignment papers...''. It's for this reason that the number of contributors (and, at second order, the success of) projects is strongly and inversely correlated with the number of hoops each project makes a user go through to contribute. Such friction costs may be political as well as mechanical. Together they may explain why the loose, amorphous Linux culture has attracted orders of magnitude more cooperative energy than the more

tightly organized and centralized BSD efforts and why the Free Software Foundation has receded in relative importance as Linux has risen.

This is all good as far as it goes. But it is an after-the-fact explanation of what J. Random Hacker does with his patch after he has it. The other half we need is an economic explanation of how JRH was able to write that patch in the first place, rather than having to work on a closed-source program that might have returned him sale value. What business models create niches in which open-source development can flourish?

6. Reasons for Closing Source

Before taxonomizing open-source business models, we should deal with exclusion payoffs in general. What exactly are we protecting when we close source?

Let's say you hire someone to write to order (say) a specialized accounting package for your business. That problem won't be solved any better if the sources are closed rather than open; the only rational reasons you might want them to be closed is if you want to sell the package to other people, or deny its use to competitors.

The obvious answer is that you're protecting sale value, but for the 95% of software written for internal use this doesn't apply. So what other gains are there in being closed?

That second case (protecting competitive advantage) bears a bit of examination. Suppose you open-source that accounting package. It becomes popular and benefits from improvements made by the community. Now, your competitor also starts to use it. The competitor gets the benefit without paying the development cost and cuts into your business. Is this an argument against open-sourcing?

Maybe -- and maybe not. The real question is whether your gain from spreading the development load exceeds your loss due to increased competition from the free rider. Many people tend to reason poorly about this tradeoff through (a) ignoring the functional advantage of recruiting more development help. (b) not treating the development costs as sunk, and By hypothesis, you had to pay th development costs anyway, so counting them as a cost of open-sourcing (if you choose to do) is mistaken.

There are other reasons for closing source that are outright irrational. You might, for example, be laboring under the delusion that closing the sources will make your business systems more secure against crackers and intruders. If so, I recommend therapeutic conversation with a cryptographer immediately. The really professional paranoids know better than to trust the security of closed-source programs, because they've learned through hard experience not to. Security is an aspect of reliability; only algorithms and implementations that have been thoroughly peer-reviewed can possibly be trusted to be secure.

7. Use-Value Funding Models

A key fact that the distinction between use and sale value allows us to notice is that only **sale value** is threatened by the shift from closed to open source; use value is not.

If use value rather than sale value is really the major driver of software development, and (as was argued in [CatB]) open-source development is really more effective and efficient than closed, then we should expect to find circumstances in which expected use value alone sustainably funds open-source development.

And in fact it is not difficult to identify at least two important models in which full-time developer salaries for open-source projects are funded strictly out of use value.

7.1 The Apache case: cost-sharing

Let's say you work for a firm that has a business-critical requirement for a high-volume, high-reliability web server. Maybe it's for electronic commerce, maybe you're a high-visibility media outlet selling advertising, maybe you're a portal site. You need 24/7 uptime, you need speed, and you need customizability.

How are you going to get these things? There are three basic strategies you can pursue:

Buy a proprietary webserver. In this case, you are betting that the vendor's agenda matches yours and that the vendor has the technical competence to implement properly. Even assuming both these things to be true, the product is likely to come up short in customizability; you will only be able to modify it through the hooks the vendor has chosen to provide. This proprietary-webserver path is not a popular one.

Roll your own. Building your own webserver is not an option to dismiss instantly; webservers are not very complex, certainly less so than browsers, and a specialized one can be very lean and mean. Going this path, you can get the exact features and customizability you want, though you'll pay for it in development time. Your firm may also find it has a problem when you retire or leave.

Join the Apache group. The Apache server was built by an Internet-connected group of webmasters who realized that it was smarter to pool their efforts into improving one code base than to run a large number of parallel development efforts. By doing this they were able to capture both most of the advantages of roll-your-own and the powerful debugging effect of massively-parallel peer review.

The advantage of the Apache choice is very strong. Just how strong, we may judge from the monthly Netcraft survey, which has shown Apache steadily gaining market share against all proprietary webservers since its inception. As of June 1999, Apache and its derivatives have 61% market share – with no legal owner, no promotion, and no contracted service organization behind them at all.

The Apache story generalizes to a model in which software users find it to their advantage to fund open-source development because doing so gets them a better product than they could otherwise have, at lower cost.

7.2 The Cisco case: risk-spreading

Some years ago, two programmers at Cisco (the networking-equipment manufacturer) got assigned the job of writing a distributed print-spooling system for use on Cisco's corporate network. This was quite a challenge. Besides supporting the ability for arbitrary user A to print at arbitrary printer B (which might be in the next room or a thousand miles away), the system had to make sure that in the event of a paper-out or toner-low condition the job would get rerouted to an alternate printer near the target. The system also needed to be able to report such problems to a printer administrator.

The duo came up with a clever set of modifications to the standard Unix print-spooler software, plus some wrapper scripts, that did the job. Then they realized that they, and Cisco, had a problem.

The problem was that neither of them was likely to be at Cisco forever. Eventually, both programmers would be gone, and the software would be unmaintained and begin to rot (that is, to gradually fall out of sync with real-world conditions). No developer likes to see this happen to his or her work, and the intrepid duo felt Cisco had paid for a solution under the not unreasonable expectation that it would outlast their own jobs there.

Accordingly, they went to their manager and urged him to authorize the release of the print spooler software as open source. Their argument was that Cisco would have no sale value to lose, and much else to gain. By encouraging the growth of a community of users and co-developers spread across many corporations, Cisco could effectively hedge against the loss of the software's original developers.

The Cisco story generalizes to a model in which open source functions not so much to lower costs as to spread risk. All parties find that the openness of the source, and the presence of a collaborative community funded by multiple independent revenue streams, provides a fail-safe that is itself economically valuable – sufficiently valuable to drive funding for it.

8. Why Sale Value is Problematic

Open source makes it rather difficult to capture direct sale value from software. The difficulty is not technical; source code is no more nor less copyable than binaries, and the enforcement of copyright and license laws permitting capture of sale value would not by necessity be any more difficult for open-source products than it is for closed.

The difficulty lies rather with the nature of the social contract that supports open-source development. For three mutually reinforcing reasons, the major open-source licenses prohibit most of the sort of restrictions on use, redistribution and modification that would facilitate direct-sale revenue capture. To understand these reasons. we must examine the social context within which the licenses evolved; the Internet hacker culture.

Despite myths about the hacker culture still (in 1999) widely believed outside it, none of these reasons has to do with hostility to the market. While a minority of hackers does indeed remain hostile to the profit motive, the general willingness of the community to cooperate with for-profit Linux packagers like Red Hat, SUSE, and Caldera demonstrates that most hackers will happily work with the corporate world when it serves their ends. The real reasons hackers frown on direct-revenue-capture licenses are more subtle and interesting.

One reason has to do with symmetry. While most open-source developers do not intrinsically object to others profiting from their gifts, most also demand that no party (with the possible exception of the originator of a piece of code) be in a **privileged** position to extract profits. J. Random Hacker is willing for Fubarco to profit by selling his software or patches, but only so long as JRH himself could also potentially do so.

Another has to do with unintended consequences. Hackers have observed that licenses that include restrictions on and fees for `commercial' use or sale (the most common form of attempt to recapture direct sale value, and not at first blush an unreasonable one) have serious chilling effects. A specific one is to cast a legal shadow on activities like redistribution in inexpensive CD-ROM anthologies, which we would ideally like to encourage. More generally, restrictions on use/sale/modification/distribution (and other complications in licensing) exact an overhead for conformance tracking and (as the number of packages people deal with rises) a combinatorial explosion of perceived uncertainty and potential legal risk. This outcome is considered harmful, and there is therefore strong social pressure to keep licenses simple and free of restrictions.

The final and most critical reason has to do with preserving the peer-review, gift-culture dynamic described in [HtN] . License restrictions designed to protect intellectual property or capture direct sale value often have the effect of making it legally impossible to fork the project (this is the case, for example, with Sun's so-called "Community Source" licenses for Jini and Java). While forking is frowned upon and considered a last resort (for reasons discussed at length in [HtN]), it's considered critically important that that last resort be present in case of maintainer incompetence or defection (e.g. to a more closed license).

The hacker community has some give on the symmetry reason; thus, it tolerates licenses like Netscape's NPL that give some profit privileges to the originators of the code (specifically in the NPL case, the exclusive right to use the open-source Mozilla code in derivative products including closed source). It has less give on the unintended-consequences reason, and none on preserving the option to fork (which is why Sun's Java and Jini `Community License' schemes have been largely rejected by the community).

These reasons explain the clauses of the Open Source Definition, which was written to express the consensus of the hacker community about the critical features of the standard licenses (the GPL, the BSD license, the MIT License, and the Artistic License). These clauses have the effect (though not the intention) of making direct sale value very hard to capture.

9. Indirect Sale-Value Models

Nevertheless, there are ways to make markets in software-related services that capture something like indirect sale value. There are five known and two speculative models of this kind (more may be developed in the future).

9.1 Loss-Leader/Market Positioner

In this model, you use open-source software to create or maintain a market position for proprietary software that generates a direct revenue stream. In the most common variant, open-source client software enables sales of server software, or subscription/advertising revenue associated with a portal site.

Netscape Communications, Inc. was pursuing this strategy when it open-sourced the Mozilla browser in early 1998. The browser side of their business was at 13% of revenues and dropping when Microsoft first shipped Internet Explorer. Intensive marketing of IE (and shady bundling practices that would later become the central issue of an antitrust lawsuit) quickly ate into Netscape's browser market share, creating concern that Microsoft intended to monopolize the browser market and then use de-facto control of HTML to drive Netscape out of the server market.

By open-sourcing the still-widely-popular Netscape browser, Netscape effectively denied Microsoft the possibility of a browser monopoly. They expected that open-source collaboration would accelerate the development and debugging of the browser, and hoped that Microsoft's IE would be reduced to playing catch-up and prevented from exclusively defining HTML.

This strategy worked. In November 1998 Netscape actually began to regain business-market share from IE. By the time Netscape was acquired by AOL in early 1999, the competitive advantage of keeping Mozilla in play was sufficiently clear that

one of AOL's first public commitments was to continue supporting the Mozilla project, even though it was still in alpha stage.

9.2 Widget Frosting

This model is for hardware manufacturers (hardware, in this context, includes anything from Ethernet or other peripheral boards all the way up to entire computer systems). Market pressures have forced hardware companies to write and maintain software (from device drivers through configuration tools all the way up to the level of entire operating systems), but the software itself is not a profit center. It's an overhead -- often a substantial one.

In this situation, opening source is a no-brainer. There's no revenue stream to lose, so there's no downside. What the vendor gains is a dramatically larger developer pool, more rapid and flexible response to customer needs, and better reliability through peer review. It gets ports to other environments for free. It probably also gains increased customer loyalty as its customers' technical staffs put increasing amounts of time into the code to do the customizations they require.

There are a couple of vendor objections commonly raised specifically to open-sourcing hardware drivers. Rather than mix them with discussion of more general issues here, I have written an appendix specifically on this topic.

The `future-proofing' effect of open source is particularly strong with respect to widget frosting. Hardware products have a finite production and support lifetime; after that, the customers are on their own. But if they have access to driver source and can patch them as needed, they're more likely to be happier repeat customers of the same company.

A very dramatic example of adopting the widget frosting model was Apple Computer's decision in mid-March 1999 to open-source "Darwin", the core of their MacOSX server operating system.

9.3 Give Away the Recipe, Open A Restaurant

In this model, one open-sources software to create a market position not for closed software (as in the Loss-Leader/Market-Positioner case) but for services.

(I used to call this `Give Away the Razor, Sell Razor Blades', The coupling is not really as tight as the razor/razor-blade analogy implies.)

This is what Red Hat and other Linux distributors do. What they are actually selling is not the software, the bits itself, but the value added by assembling and testing a running operating system that is warranted (if only implicitly) to be merchantable and to be plug-compatible with other operating systems carrying the same brand. Other elements of their value proposition include free installation support and the provision of options for continuing support contracts.

The market-building effect of open source can be extremely powerful, especially for companies which are inevitably in a service position to begin with. One very instructive recent case is Digital Creations, a website-design house started up in 1998 that specializes in complex database and transaction sites. Their major tool, the intellectual-property crown jewels of the company, is an object publisher that has been through several names and incarnations but is now called Zope.

When the Digital Creations people went looking for venture capital, the VC they brought in carefully evaluated their prospective market niche, their people, and their tools. He then recommended that Digital Creations take Zope to open source.

By traditional software-industry standards, this looks like an absolutely crazy move. Conventional business-school wisdom has it that core intellectual property like Zope is a company's crown jewels, never under any circumstances to be given away. But the VC had two related insights. One is that Zope's true core asset is actually the brains and skills of its people. The second is that Zope is likely to generate more value as a market-builder than as a secret tool.

To see this, compare two scenarios. In the conventional one, Zope remains Digital Creations's secret weapon. Let's stipulate that it's a very effective one. As a result, the firm will able to deliver superior quality on short schedules -- **but nobody knows that**. It will be easy to satisfy customers, but harder to build a customer base to begin with.

The VC, instead, saw that open-sourcing Zope could be critical advertising for Digital Creations's **real** asset -- its people. He expected that customers evaluating Zope would consider it more efficient to hire the experts than to develop in-house Zope expertise.

One of the Zope principals has since confirmed very publicly that their open-source strategy has "opened many doors we wouldn't have got in otherwise". Potential customers do indeed respond to the logic of the situation -- and Digital Creations, accordingly, is prospering.

Another up-to-the-minute example is e-smith, inc.. This company sells support contracts for turnkey Internet server software that is open-source, a customized Linux. One of the principals, describing the spread of free downloads of e-smith's software, says ``Most companies would consider that software piracy; we consider it free marketing''.

9.4 Accessorizing

In this model, you sell accessories for open-source software. At the low end, mugs and T-shirts; at the high end, professionally-edited and produced documentation.

O'Reilly Associates, publishers of many excellent references volumes on open-source software, is a good example of an accessorizing company. O'Reilly actually hires and supports well-known open-source hackers (such as Larry Wall and Brian Behlendorf) as a way of building its reputation in its chosen market.

9.5 Free the Future, Sell the Present

In this model, you release software in binaries and source with a closed license, but one that includes an expiration date on the closure provisions. For example, you might write a license that permits free redistribution, forbids commercial use without fee, and guarantees that the software come under GPL terms a year after release or if the vendor folds.

Under this model, customers can ensure that the product is customizable to their needs, because they have the source. The product is future-proofed -- the license guarantees that an open source community can take over the product if the original company dies.

Because the sale price and volume are based on these customer expectations, the original company should enjoy enhanced revenues from its product versus releasing it with an exclusively closed source license. Furthermore, as older code is GPLed, it will get serious peer review, bug fixes, and minor features, which

removes some of the 75% maintainance burden on the originator.

This model has been successfully pursued by Aladdin Enterprises, makers of the popular Ghostscript program (a PostScript interpreter that can translate to the native languages of many printers).

The main drawback of this model is that the closure provisions tend to inhibit peer review and participation early in the product cycle, precisely when they are needed most.

9.6 Free the Software, Sell the Brand

This is a speculative business model. You open-source a software technology, retain a test suite or set of compatibility criteria, then sell users a brand certifying that their implementation of the technology is compatible with all others wearing the brand.

(This is how Sun Microsystems ought to be handling Java and Jini.)

9.7 Free the Software, Sell the Content

This is another speculative business model. Imagine something like a stock-ticker subscription service. The value is neither in the client software nor the server but in providing objectively reliable information. So you open-source all the software and sell subscriptions to the content. As hackers port the client to new platforms and enhance it in various ways, your market automatically expands.

(This is why AOL ought to open-source its client software.)

10. When To Be Open, When To Be Closed

Having reviewed business models that support open-source software development, we can now approach the general question of when it makes economic sense to be open-source and when to be closed-source. First, we must be clear what the payoffs are from each strategy.

10.1 What Are the Payoffs?

The closed-source approach allows you to collect rent from your secret bits; on the other hand, it forecloses the possibility of truly independent peer review. The open-source approach sets up conditions for independent peer review, but you don't get rent from your secret bits.

The payoff from having secret bits is well understood; traditionally, software business models have been constructed around it. Until recently, the payoff from independent peer review was not well understood. The Linux operating system, however, drives home a lesson that we should probably have learned years ago from the history of the Internet's core software and other branches of engineering -- that open-source peer review is the only scalable method for achieving high reliability and quality.

In a competitive market, therefore, customers seeking high reliability and quality will reward software producers who go open-source and discover how to maintain a revenue stream in the service, value-add, and ancilliary markets associated with software. This phenomenon is what's behind the astonishing success of Linux, which came from nowhere in 1996 to over 17% in the business server market by the end of 1998 and seems on track to dominate that market within two years (in early 1999 IDC projected that Linux would grow faster than all other operating systems combined through 2003).

An almost equally important payoff of open source is its utility as a way to propagate open standards and build markets around them. The dramatic growth of the Internet owes much to the fact that nobody owns TCP/IP; nobody has a proprietary lock on the core Internet protocols.

The network effects behind TCP/IP's and Linux's success are fairly clear and reduce ultimately to issues of trust and symmetry -- potential parties to a shared infrastructure can rationally trust it more if they can see how it works all the way down, and will prefer an infrastructure in which all parties have symmetrical rights to one in which a single party is in a privileged position to extract rents or exert control.

It is not, however, actually necessary to assume network effects in order for symmetry issues to be important to software consumers. No software consumer will rationally choose to lock itself into a supplier-controlled monopoly by becoming dependent on closed source if any open-source alternative of acceptable quality is available. This argument gains force as the software becomes more critical to the software consumer's business – the more vital it is, the less the consumer can tolerate having it controlled by an outside party.

Finally, an important customer payoff of open-source software related to the trust issue is that it's future-proof. If sources are open, the customer has some recourse if the vendor goes belly-up. This may be particularly important for widget frosting, since hardware tends to have short life cycles, but the effect is more general and translates into increased value for open-source software.

10.2 How Do They Interact?

When the rent from secret bits is higher than the return from open source, it makes economic sense to be closed-source. When the return from open source is higher than the rent from secret bits, it makes sense to go open source.

In itself, this is a trivial observation. It becomes nontrivial when we notice that the payoff from open source is harder to measure and predict than the rent from secret bits -- and that said payoff is grossly underestimated much more often than it is overestimated. Indeed, until the mainstream business world began to rethink its premises following the Mozilla source release in early 1998, the open-source payoff was incorrectly but very generally assumed to be zero.

So how can we evaluate the payoff from open source? It's a difficult question in general, but we can approach it as we would any other predictive problem. We can start from observed cases where the open-source approach has succeeded or failed. We can try to generalize to a model which gives at least a qualitative feel for the contexts in which open source is a net win for the investor or business trying to maximize returns. We can then go back to the data and try to refine the model.

From the analysis presented in [CatB] , we can expect that open source has a high payoff where (a) reliability/stability/scalability are critical, and (b) correctness of design and implementation is not readily verified by means other than independent peer review. (The second criterion is met in practice by most non-trivial programs.)

A consumer's rational desire to avoid being locked into a monopoly supplier will increase its interest in open source (and, hence, the competitive-market value for suppliers of going open) as the software becomes more critical to that consumer. Thus, another criterion (c) pushes towards open source when the

software is a business-critical capital good (as, for example, in many corporate MIS departments).

As for application area, we observed above that open-source infrastructure creates trust and symmetry effects that, over time, will tend to attract more customers and to outcompete closed-source infrastructure; and it is often better to have a smaller piece of such a rapidly-expanding market than a bigger piece of a closed and stagnant one. Accordingly, for infrastructure software, an open-source play for ubiquity is quite likely to have a higher long-term payoff than a closed-source play for rent from intellectual property.

In fact, the ability of potential customers to reason about the future consequences of vendor strategies and their reluctance to accept a supplier monopoly implies a stronger constraint; without already having overwhelming market power, you can choose either an open-source ubiquity play or a direct-revenue-from-closed-source play -- but not both. (Analogues of this principle are visible elsewhere, e.g. in electronics markets where customers often refuse to buy sole-source designs.) The case can be put less negatively: where network effects (positive network externalities) dominate, open source is likely to be the right thing.

We may sum up this logic by observing that open source seems to be most successful in generating greater returns than closed source in software that (d) establishes or enables a common computing and communications infrastructure.

Finally, we may note that purveyors of unique or just highly differentiated services have more incentive to fear copying of their methods by competitors than do vendors of services for which the critical algorithms and knowledge bases are well understood. Accordingly, open source is more likely to dominate when (e) key methods (or functional equivalents) are part of common engineering knowledge.

The Internet core software, Apache, and Linux's implementation of the ANSI-standard Unix API are prime exemplars of all five criteria. The path towards open source in the evolution of such markets are well-illustrated by the reconvergence of data networking on TCP/IP in the mid-1990s following fifteen years of failed empire-building attempts with closed protocols such as DECNET, XNS, IPX, and the like.

On the other hand, open source seems to make the least sense for companies that have unique possession of a value-generating software technology (strongly fulfilling criterion (e)) which is (a) relatively insensitive to failure, which can (b) readily be verified by means other than independent peer review, which is not (c) business-critical, and which would not have its value substantially increased by (d) network effects or ubiquity.

As an example of this extreme case, in early 1999 I was asked "Should we go open source?" by a company that writes software to calculate cutting patterns for sawmills that want to extract the maximum yardage of planks from logs. My conclusion was ``No.'' The only criterion this comes even close to fulfilling is (c); but at a pinch, an experienced operator could generate cut patterns by hand.

An important point is that where a particular product or technology sits on these scales may change over time, as we'll see in the following case study.

In summary, the following discriminators push towards open source:

(a)
reliability/stability/scalability are critical

(b)
correctness of design and implementation cannot readily be verified by means other than independent peer review

(c)
the software is critical to the user's control of his/her business

(d)
the software establishes or enables a common computing and communications infrastructure

(e)
key methods (or functional equivalents of them) are part of common engineering knowledge.

10.3 Doom: A Case Study

The history of id software's best-selling game Doom illustrates ways in which market pressure and product evolution can critically change the payoff magnitudes for closed vs. open source.

When Doom was first released in late 1993, its first-person, real-time animation made it utterly unique (the antithesis of criterion (e)). Not only was the visual impact of the technique stunning, but for many months nobody could figure out how it had been achieved on the underpowered microprocessors of that time. These secret bits were worth some very serious rent. In addition, the potential payoff from open source was low. As a solo game, the software (a) incurred tolerably low costs on failure, (b) not tremendously hard to verify, (c) not business-critical for any consumer, (d) did not benefit from network effects. It was economically rational for Doom to be closed source.

However, the market around Doom did not stand still. Would-be competitors invented functional equivalents of its animation techniques, and other ``first-person shooter'' games like Duke Nukem began to appear. As these games ate into Doom's market share the value of the rent from secret bits went down.

On the other hand, efforts to expand that share brought on new technical challenges -- better reliability, more game features, a larger user base, and multiple platforms. With the advent of multiplayer `deathmatch' play and Doom gaming services, the market began to display substantial network effects. All this was demanding programmer-hours that id would have preferred to spend on the next game.

All of these trends raised the payoff from opening the source. At some point the payoff curves crossed over and it became economically rational for id to open up the Doom source and shift to making money in secondary markets such as game-scenario anthologies. And sometime after this point, it actually happened. The full source for Doom was released in late 1997.

10.4 Knowing When To Let Go

Doom makes an interesting case study because it is neither an operating system nor communications/networking software; it is thus far removed from the usual and obvious examples of open-source success. Indeed, Doom's life cycle, complete with crossover point, may be coming to typify that of applications software in today's code ecology -- one in which communications and distributed computation both create serious robustness/reliability/scalability problems only addressible by peer review, and frequently cross boundaries both between technical environments and between competing actors (with all the trust and symmetry issues that implies).

Doom evolved from solo to deathmatch play. Increasingly, the network effect **is** the computation. Similar trends are visible even

in the heaviest business applications, such as ERPs, as businesses network ever more intensively with suppliers and customers -- and, of course, they are implicit in the whole architecture of the World Wide Web. It follows that almost everywhere, the open-source payoff is steadily rising.

If present trends continue, the central challenge of software technology and product management in the next century will be knowing when to let go – when to allow closed code to pass into the open-source infrastructure in order to exploit the peer-review effect and capture higher returns in service and other secondary markets.

There are obvious revenue incentives not to miss the crossover point too far in either direction. Beyond that, there's a serious opportunity risk in waiting too long -- you could get scooped by a competitor going open-source in the same market niche.

The reason this is a serious issue is that both the pool of users and the pool of talent available to be recruited into open-source cooperation for any given product category is limited, and recruitment tends to stick. If two producers are the first and second to open-source competing code of roughly equal function, the first is likely to attract the most users and the most and best-motivated co-developers; the second will have to take leavings. Recruitment tends to stick, as users gain familiarity and developers sink time investments in the code itself.

11. The Business Ecology of Open Source

The open-source community has organized itself in a way that tends to amplify the productivity effects of open source. In the Linux world, in particular, it's an economically significant fact that there are multiple competing Linux distributors which form a tier separate from the developers.

Developers write code, and make the code available over the Internet. Each distributor selects some subset of the available code, integrates and packages and brands it, and sells it to customers. Users choose among distributions, and may supplement a distribution by downloading code directly from developer sites.

The effect of this tier separation is to create a very fluid internal market for improvements. Developers compete with each other, for the attention of distributors and users, on the quality of their software. Distributors compete for user dollars on the appropriateness of their selection policies, and on the value they can add to the software.

A first-order effect of this internal market structure is that no node in the net is indispensible. Developers can drop out; even if their portion of the code base is not picked up directly by some other developer, the competition for attention will tend to rapidly generate functional alternatives. Distributors can fail without damaging or compromising the common open-source code base. The ecology as a whole has a more rapid response to market demands, and more capability to resist shocks and regenerate itself, than any monolithic vendor of a closed-source operating system can possibly muster.

Another important effect is to lower overhead and increase efficiency through specialization. Developers don't experience the pressures that routinely compromise conventional closed projects and turn them into tar-pits -- no lists of pointless and distracting check-list features from Marketing, no management mandates to use inappropriate and outdated languages or development environments, no requirement to re-invent wheels in a new and incompatible way in the name of product differentiation or intellectual-property protection, and (most importantly) **no deadlines**. No rushing a 1.0 out the door before it's done right -- which (as DeMarco and Lister observed in their discussion of the `wake me when it's over' management style in [DL]) generally conduces not only to higher quality but actually to the most rapid delivery of a truly working result.

Distributors, on the other hand, get to specialize in the things distributors can do most effectively. Freed of the need to fund massive and ongoing software development just to stay competitive, they can concentrate on system integration, packaging, quality assurance, and service.

Both distributors and developers are kept honest by the constant feedback from and monitoring by users that is an integral part of the open-source method.

12. Coping With Success

The Tragedy of the Commons may not be applicable to open-source development as it happens today, but that doesn't mean there are not any reasons to wonder if the present momentum of the open-source community is sustainable. Will key players defect from cooperation as the stakes become higher?

There are several levels on which this question can be asked. Our `Comedy of the Commons' counter-story is based on the argument that the value of individual contributions to open source is hard to monetize. But this argument has much less force for firms (like, say, Linux distributors) which already have a revenue stream associated with open source. Their contribution is already being monetized every day. Is their present cooperative role stable?

Examining this question will lead us to some interesting insights about the economics of open-source software in the real world of present time -- and about what a true service-industry paradigm implies for the software industry in the future.

On the practical level, applied to the open-source community as it exists now, this question is usually posed in one of two different ways. One: will Linux fragment? Two: conversely, will Linux develop a dominant, quasi-monopolistic player?

The historical analogy many people turn to when suggesting that Linux will fragment is the behavior of the proprietary-Unix vendors in the 1980s. Despite endless talk of open standards, despite numerous alliances and consortia and agreements, proprietary Unix fell apart. The vendors' desire to differentiate their products by adding and modifying OS facilities proved stronger than their interest in growing the total size of the Unix market by maintaining compatibility (and consequently lowering both entry barriers for independent software developers and total cost of ownership for consumers).

This is quite unlikely to happen to Linux, for the simple reason that all the distributors are constrained to operate from a common base of open source code. It's not really possible for any one of them to maintain differentiation, because the licenses under which Linux code are developed effectively require them to share code with all parties. The moment any distributor develops a feature, all competitors are free to clone it.

Since all parties understand this, nobody even thinks about doing the kinds of maneuvers that fragmented proprietary Unix. Instead, Linux distributors are forced to compete in ways that actually **benefit** the consumer and the overall market. That is, they must compete on service, support, and their design bets on what interfaces actually conduce to ease installation and use.

The common source base also forecloses the possibility of monopolization. When Linux people worry about this, the name usually muttered is ``Red Hat'', that of the largest and most successful of the distributors (with somewhere around 90% estimated market share in the U.S.). But it is notable that within days after the May 1999 announcement of Red Hat's long-awaited 6.0 release -- before Red Hat's CD-ROMs actually shipped in any quantity – CD-ROM images of the release built from Red Hat's own public FTP site were being advertised by a book publisher and several other CD-ROM distributors at lower prices than Red Hat's expected list.

Red Hat itself didn't turn a hair at this, because its founders understand very clearly that they do not and cannot own the bits in their product; the social norms of the Linux community forbid that. In a latter-day take on John Gilmore's famous observation that the Internet interprets censorship as damage and routes around it, it has been aptly said that the hacker community responsible for Linux interprets attempts at control as damage and routes around them. For Red Hat to have protested the pre-release cloning of its newest product would have seriously compromised its ability to elicit future cooperation from its developer community.

Perhaps more importantly in present time, the software licenses that express these community norms in a binding legal form actively forbid Red Hat from monopolizing the sources of the code their product is based on. The only thing they can sell is a brand/service/support relationship with people who are freely willing to pay for that. This is not a context in which the possibility of a predatory monopoly looms very large.

13. Open R&D and the Reinvention of Patronage

There is one other respect in which the infusion of real money into the open-source world is changing it. The community's stars are increasingly finding they can get paid for what they want to do, instead of pursuing open source as a hobby funded by another day job. Corporations like Red Hat, O'Reilly Associates, and VA Linux Systems are building what amount to semi-independent research arms with charters to hire and maintain stables of open-source talent.

This makes economic sense only if the cost per head of maintaining such a lab can easily be paid out of the expected gains it will enable by growing the firm's market faster. O'Reilly can afford to pay the principal authors of Perl and Apache to do their thing because it expects their efforts will enable it to sell more Perl- and Apache-related books. VA Linux Systems can fund its laboratory branch because improving Linux boosts the use value of the workstations and servers it sells. And Red Hat funds Red Hat Advanced Development Labs to increase the value of its Linux offering and attract more customers.

To strategists from more traditional sectors of the software industry, reared in cultures that regard patent- or trade-secret-protected intellectual property as the corporate crown jewels, this behavior may (despite its market-growing effect) seem inexplicable. Why fund research that every one of your competitors is (by definition) free to appropriate at no cost?

There seem to be two controlling reasons. One is that as long as these companies remain dominant players in their market niches, they can expect to capture a proportional lion's share of the returns from the open R&D. Using R&D to buy future profits is hardly a novel idea; what's interesting is the implied calculation that the expected future gains are sufficiently large that these companies can readily tolerate free riders.

While this obvious expected-future-value analysis is a necessary one in a world of hard-nosed capitalists keeping their eyes on ROI, it is not actually the most interesting mode of explanation for star-hiring, because the firms themselves advance a fuzzier one. They will tell you if asked that they are simply doing the right thing by the community they come from. Your humble author is sufficiently well-acquainted with principals at all three of the firms cited above to testify that these protestations cannot be dismissed as humbug. Indeed, I was personally recruited onto the board of VA Linux Systems in late 1998 explicitly so that I would be available to advise them on ``the right thing'', and have found them far from unwilling to listen when I did so.

An economist is entitled to ask what payoff is involved here. If we accept that talk of doing the right thing is not empty posturing, we should next inquire what self-interest of the firm the "right thing" serves. Nor is the answer, in itself, either surprising or difficult to verify by asking the right questions. As with superficially altruistic behavior in other industries, what these firms actually believe they're buying is goodwill.

Working to earn goodwill, and valuing it as an asset predictive of future market gains, is hardly novel either. What's interesting is the extremely high valuation that the behavior of these firms suggest they put on that goodwill. They're demonstrably willing to hire expensive talent for projects that are not direct revenue generators even during the most capital-hungry phases of the runup to IPO. And, at least so far, the market has actually rewarded this behavior.

The principals of these companies themselves are quite clear about the reasons that goodwill is especially valuable to them. They rely heavily on volunteers among their customer base both for product development and as an informal marketing arm. Their relationship with their customer base is intimate, often relying on personal trust bonds between individuals within and outside the firm.

These observations reinforce a lesson we learned earlier from a different line of reasoning. The relationship between Red Hat/VA/O'Reilly and their customers/developers is not one typical of manufacturing firms. Rather, it carries to an interesting extreme patterns that are characteristic of knowledge-intensive service industries. Looking outside the technology industry, we can see these patterns in (for example) law firms, medical practices, and universities.

We may observe, in fact, that open-source firms hire star hackers for much the same reasons that universities hire star academics. In both cases, the practice is similar in mechanism and effect to the system of aristocratic patronage that funded most fine art until after the Industrial Revolution -- a similarity some parties to it are fully aware of.

14. Getting There From Here

The market mechanisms for funding (and making a profit from!) open-source development are still evolving rapidly. The business models we've reviewed in this paper probably will not be the last to be invented. Investors are still thinking through the consequences of reinventing the software industry as one with an explicit focus on service rather than closed intellectual property, and will be for some time to come.

This conceptual revolution will have some cost in foregone profits for people investing in the sale-value 5% of the industry; historically, service businesses are not as lucrative as

manufacturing businesses (though as any doctor or lawyer could tell you, the return to the actual practitioners is often higher). Any foregone profits, however, will be more than matched by benefits on the cost side, as software consumers reap tremendous savings and efficiencies from open-source products. (There's a parallel here to the effects that the displacement of the traditional voice-telephone network by the Internet is having everywhere).

The promise of these savings and efficiencies is creating a market opportunity that entrepreneurs and venture capitalists are now moving in to exploit. As the first draft of this paper was in preparation, Silicon Valley's most prestigious venture-capital firm took a lead stake in the first startup company to specialize in 24/7 Linux technical support. It is generally expected that several Linux- and open-source-related IPOs will be floated before the end of 1999 -- and that they will be quite successful.

Another very interesting development is the beginnings of systematic attempts to make task markets in open-source development. SourceXchange and CoSource represent slightly different ways of trying to apply a reverse-auction model to funding open-source development.

The overall trends are clear. We mentioned before IDC's projection that Linux will grow faster than all other operating systems **combined** through 2003. Apache is at 61% market share and rising steadily. Internet usage is exploding, and surveys such as the Internet Operating System Counter show that Linux and other open-source operating systems are already a plurality on Internet hosts and steadily gaining share against closed systems. The need to exploit open-source Internet infrastructure increasingly conditions not merely the design of other software but the business practices and software use/purchase patterns of every corporation there is. These trends, if anything, seem likely to accelerate.

15. Conclusion: Life After The Revolution

What will the world of software look like once the open-source transition is complete?

For purposes of examining this question, it will be helpful to sort kinds of software by the degree of completeness which the service they offer is describable by open technical standards, which is well correlated with how commoditized the underlying service has become.

This axis corresponds reasonably well to what people are normally thinking when they speak of `applications' (not at all commoditized, weak or nonexistent open technical standards), `infrastructure' (commoditized services, strong standards), and `middleware' (partially commoditized, effective but incomplete technical standards). The paradigm cases today in 1999 would be a word processor (application), a TCP/IP stack (infrastructure), and a database engine (middleware).

The payoff analysis we did earlier suggests that infrastructure, applications, and middleware will be transformed in different ways and exhibit different equilibrium mixes of open and closed source. We recall that it also suggested the prevalence of open source in a particular software area would be a function of whether substantial network effects operate there, what the costs of failure are, and to what extent the software is a business-critical capital good.

We can venture some predictions if we apply these heuristics not to individual products but to entire segments of the software market. Here we go:

Infrastructure (the Internet, the Web, operating systems, and the lower levels of communications software that has to cross boundaries between competing parties) will be almost all open source, cooperatively maintained by user consortia and by for-profit distribution/service outfits with a role like that of Red Hat today.

Applications, on the other hand, will have the most tendency to remain closed. There will be circumstances under which the use value of an undisclosed algorithm or technology will be high enough (and the costs associated with unreliability will be low enough, and the risks associated with a supplier monopoly sufficiently tolerable) that consumers will continue to pay for closed software. This is likeliest to remain true in standalone vertical-market applications where network effects are weak. Our lumber-mill example earlier is one such; biometric identification software seems likeliest, of 1999's hot prospects, to be another.

Middleware (like databases, development tools, or the customized top ends of application protocol stacks) will be more mixed. Whether middleware categories tend to go closed or open seems likely to depend on the cost of failures, with higher cost creating market pressure for more openness.

To complete the picture, however, we need to notice that neither `applications' nor `middleware' are really stable categories. In `Knowing When To Let Go' above we saw that individual software technologies seem to go through a natural life cycle from rationally closed to rationally open. The same logic applies in the large.

Applications tend to fall into middleware as standardized techniques develop and portions of the service becomes commoditized. (Databases, for example, became middleware after SQL decoupled front ends from engines.) As middleware services become commoditized, they will in turn tend to fall into the open-source infrastructure -- a transition we're seeing in operating systems right now.

In a future that includes competition from open source, we can expect that the eventual destiny of any software technology will be to either die or become part of the open infrastructure itself. While this is hardly happy news for entrepreneurs who would like to collect rent on closed software forever, it does suggest that the software industry as a whole will **remain** entrepreneurial, with new niches constantly opening up at the upper (application) end and a limited lifespan for closed-IP monopolies as their product categories fall into infrastructure.

Finally, of course, this equilibrium will be great for the software consumer driving the process. More and more high-quality software will become permanently available to use and build on instead of being discontinued or locked in somebody's vault. Ceridwen's magic cauldron is, finally, too weak a metaphor -- because food is consumed or decays, whereas software sources potentially last forever. The free market, in its widest libertarian sense including **all** un-coerced activity whether trade or gift, can produce perpetually increasing software wealth for everyone.

16. Bibliography and Acknowledgements

[CatB] The Cathedral and the Bazaar

[HtN] Homesteading the Noosphere

[DL] De Marco and Lister, **Peopleware: Productive Projects and Teams** (New York; Dorset House, 1987; ISBN 0-932633-05-6)

[SH] Shawn Hargreaves has written a good analysis of the applicability of open-source methods to games; Playing the Open Source Game.

Several stimulating discussions with David D. Friedman helped me refine the `inverse commons' model of open-source cooperation. I am also indebted to Marshall van Alstyne for pointing out the conceptual importance of rivalrous information goods. Ray Ontko of the Indiana Group supplied helpful criticism. A good many people in audiences before whom I gave talks in the year leading up to June 1999 also helped; if you're one of those, you know who you are.

It's yet another testimony to the open-source model that this paper was substantially improved by email feedback I received within days after release. Lloyd Wood pointed out the importance of open-source software being `future-proof'. and Doug Dante reminded m e of the `Free the Future' business model. A question from Adam Moorhouse led to the discussion of exclusion payoffs. Lionel Oliviera Gresse gave me a better name for one of the business models. Stephen Turnbull slapped me silly about careless handling of free-rider effects.

17. Appendix: Why Closing Drivers Loses A Vendor Money

Manufacturers of peripheral hardware (Ethernet cards, disk controllers, video board and the like) have historically been reluctant to open up. This is changing now, with players like Adaptec and Cyclades beginning to routinely disclose specifications and driver source code for their boards. Nevertheless, therere's still resistance out there. In this appendix we attempt to dispel several ofthe economic misconceptions that sustain it.

If you are a hardware vendor, you may fear open-sourcing may reveal important things about how your hardware operates that competitors could copy, thus gaining an unfair competitive advantage. Back in the days of three- to five-year product cycles this was a valid argument. Today, the time your competitors' engineers would need to spend copying and understanding the copy is a substantial portion of the product cycle, time they are **not** spending innovating or differentiating their own product. Plagiarism is a trap you **want** your competitors to fall into.)

In any case, these details don't stay hidden for long these days. Hardware drivers are not like operating systems or applications; they're small, easy to disassemble, and easy to clone. Even teenage novice programmers can do this -- and frequently do.

There are literally thousands of Linux and FreeBSD programmers out there with both the capability and the motivation to build drivers for a new board. For many classes of device that have relatively simple interfaces and well-known standards (such as disk controllers and network cards) these eager hackers can often prototype a driver as almost rapidly as your own shop could, even without documentation and without disassembling an existing driver.

Even for tricky devices like video cards, there is not much you can do to thwart a clever programmer armed with a disassembler. Costs are low and legal barriers are porous; Linux is an international effort and there is always a jurisdiction in which reverse-engineering will be legal.

For hard evidence that all these claims are true, examine the list of devices supported in the Linux kernel or in the driver subtrees of sites like Metalab, and notice the rate at which new ones are added.

The message? Keeping your driver secret looks attractive in the short run, but is probably bad strategy in the long run (certainly when you're competing with other vendors that are already open). But if you must do it, burn the code into an onboard ROM. Then publish the interface to it. Go open as much as possible, to build your market and demonstrate to potentional customers that you believe in your capacity to out-think and out-innovate competitors where it matters.

If you stay closed you will usually get the worst of all worlds -- your secrets will get exposed, you won't get free development help, and you won't have wasted your stupider competition's time on cloning. Most importantly, you miss an avenue to widespread early adoption. A large and influential market (the people who manage the servers that run effectively all of the Internet and more than 17% of business data centers) will correctly write your company off as clueless and defensive because you didn't realize these things. Then they'll buy their boards from someone who did.

Halloween I

COLOR CODING NOTE:
{green} = comments by Eric Raymond
red = original text, highlighted by Eric Raymond
black = original text

Vinod Valloppillil (VinodV)
Aug 11, 1998 -- v1.00
Microsoft Confidential

Open Source Software

A (New?) Development Methodology

Executive Summary

Open Source Software (OSS) is a development process which promotes rapid creation and deployment of incremental features and bug fixes in an existing code / knowledge base. In recent years, corresponding to the growth of Internet, OSS projects have acquired the depth & complexity traditionally associated with commercial projects such as Operating Systems and mission critical servers.

Consequently, OSS poses a direct, short-term revenue and platform threat to Microsoft – particularly in server space. Additionally, the intrinsic parallelism and free idea exchange in OSS has benefits that are not replicable with our current licensing model and therefore present a long term developer mindshare threat.

{ OK, this establishes that Microsoft isn't asleep at the switch.

TN explains the connection to Java as follows:
Okay, what does this basically mean? Microsoft perceives a product to be a ``threat'' if it presents itself as any of these:

1. a revenue alternative -- somebody might spend money on a non-MS -- product
2. a platform alternative -- MS might lose its monopoly position
3. a developer alternative -- people might actually write software for a non-MS product. In their minds, any alternative is a threat. Therefore, freedom of choice is a

source of fear and loathing to MS. The idea that there may be zero (or negative!) costs with leaving MS and migrating to another platform scares the daylights out of MS. }

However, other OSS process weaknesses provide an avenue for Microsoft to garner advantage in key feature areas such as architectural improvements (e.g. storage+), integration (e.g. schemas), ease-of-use, and organizational support.

{ This summary recommendation is mainly interesting for how it fails to cover the specific suggestions later on in the document about de-commoditizing protocols etc. I'm told by a former Microserf that the references to "Storage+" here and in the executive summary are much more significant than they seem. MS's plan for the next few years is to move to an integrated file/data/storage system based upon Exchange, completely replacing the current FAT and NTFS file systems. They are absolutely planning on one monolithic structure, called "megaserver", as their next strategic infrastructure. The lock-in effect of this would be immense if they succeed. }

Open Source Software

What is it?

Open Source Software (OSS) is software in which both source and binaries are distributed or accessible for a given product, usually for free. OSS is often mistaken for "shareware" or "freeware" but there are significant differences between these licensing models and the process around each product.

Software Licensing Taxonomy

Software Type							
Commercial							
Trial Software	X (Non-full featured)	X					
Non-Commercial Use	X (Usage dependent)	X					
Shareware	X (Unenforced licensing)	X					
Royalty-free binaries ("Freeware")	X	X	X				
Royalty-free libraries	X	X	X	X			
Open Source (BSD-Style)	X	X	X	X	X		
Open Source (Apache Style)	X	X	X	X	X	X	
Open Source (Linux/GNU style)	X	X	X	X	X	X	X
License Feature	**Zero Price Avenue**	**Redistributable**	**Unlimited Usage**	**Source Code Available**	**Source Code Modifiable**	**Public "Check-ins" to core codebase**	**All derivatives must be free**

The broad categories of licensing include:

- **Commercial software**

Commercial software is classic Microsoft bread-and-butter. It must be purchased, may NOT be redistributed, and is typically only available as binaries to end users.

- **Limited trial software**

Limited trial software are usually functionally limited versions of commercial software which are freely distributed and intend to drive purchase of the commercial code. Examples include 60-day time bombed evaluation products.

- **Shareware**

Shareware products are fully functional and freely redistributable but have a license that mandates eventual purchase by both individuals and corporations. Many internet utilities (like "WinZip") take advantage of shareware as a distribution method.

- **Non-commercial use**

Non-commercial use software is freely available and redistributable by non-profit making entities. Corporations, etc. must purchase the product. An example of this would be Netscape Navigator.

- **Royalty free binaries**

Royalty-free binaries consist of software which may be freely used and distributed in binary form only. Internet Explorer and NetMeeting binaries fit this model.

- **Royalty free libraries**

Royalty-free libraries are software products whose binaries and source code are freely used and distributed but may NOT be modified by the end customer without violating the license. Examples of this include class libraries, header files, etc.

- **Open Source (BSD-style)**

A small, closed team of developers develops BSD-style open source products & allows free use and redistribution of binaries and code. While users are allowed to modify the code, the development team does NOT typically take "check-ins" from the public.

- **Open Source (Apache-style)**

Apache takes the BSD-style open source model and extends it by allowing check-ins to the core codebase by external parties.

- **Open Source (CopyLeft, Linux-style)**

CopyLeft or GPL (General Public License) based software takes the Open Source license one critical step farther. Whereas BSD and Apache style software permits users to "fork" the codebase and apply their own license terms to their modified code (e.g. make it commercial), the GPL license requires that all derivative works in turn must also be GPL code. "You are free to hack this code as long as your derivative is also hackable"

{ It's interesting to note how differently these last three distinctions are framed from the way the open-source community generally views them.

To us, open-source licensing and the rights it grants to users and third parties are primary, and specific development practice varies ad-hoc in a way not especially coupled to our license variations. In this Microsoft taxonomy, on the other hand, the central distinction is who has write access to a privileged central code base.

This reflects a much more centralized view of reality, and reflects a failure of imagination or understanding on the memo-authors's part. He doesn't grok our distributed-development tradition fully. This is hardly surprising... }

Open Source Software is Significant to Microsoft

This paper focuses on Open Source Software (OSS). OSS is acutely different from the other forms of licensing (in particular "shareware") in two very important respects:

1. There always exists an avenue for completely royalty-free purchase of the core code base

2. Unlike freely distributed binaries, Open Source encourages a *process* around a core code base and encourages extensions to the codebase by other developers.

OSS is a concern to Microsoft for several reasons:

1. OSS projects have achieved "commercial quality"

A key barrier to entry for OSS in many customer environments has been its perceived lack of quality. OSS advocates contend that the greater code inspection & debugging in OSS software results in higher quality code than commercial software.

Recent case studies (the Internet) provide very dramatic evidence in customer's eyes that commercial quality can be achieved / exceeded by OSS projects. At this time, however there is no strong evidence of OSS code quality aside from anecdotal.

{ These sentences, taken together, are rather contradictory unless the ``recent case studies'' are all ``anecdotal''. But if so, why call them ``very dramatic evidence''?

It appears there's a bit of self-protective backing and filling going on in the second sentence. Nevertheless, the first sentence is a huge concession for Microsoft to make (even internally).

In any case, the `anecdotal' claim is false. See Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services .

Here are three pertinent lines from this paper:

"The failure rate of utilities on the commercial versions of UNIX that we tested . . . ranged from 15-43%." "The failure rate of the utilities on the freely-distributed Linux version of UNIX was second-lowest, at 9%." "The failure rate of the public GNU utilities was the lowest in our study, at only 7%.

TN remarks:
Note the clever distinction here (which Eric missed in his analysis). ``customer's eyes'' (in Microsoft's own words) rather than any **real** code quality. In other words, to Microsoft and the software market in general, a software product has "commercial quality" if it has the ``look and feel'' of commercial software products. A product has commercial quality code if and only if there is a **public perception** that it is made with commercial quality code. This means that MS will take seriously any product that has an appealing, commercial-looking appearance because MS assumes -- rightly so -- that this is what the typical, uninformed consumer uses as the judgment benchmark for what is ``good code''.

TN is probably right. This didn't occur to me because, like most open-source programmers , I consider programs that crash and screw up a lot to be junk no matter how pretty their interfaces are....

}

2. OSS projects have become large-scale & complex

Another barrier to entry that has been tackled by OSS is project complexity. OSS teams are undertaking projects whose size & complexity had heretofore been the exclusive domain of commercial, economically-organized/motivated development teams. Examples include the Linux Operating System and Xfree86 GUI.

OSS process vitality is directly tied to the Internet to provide distributed development resources on a mammoth scale. Some examples of OSS project size:

Project	Lines of Code
Linux Kernel (x86 only)	500,000
Apache Web Server	80,000
SendMail	57,000

Xfree86 X-windows server	1.5 Million
"K" desktop environment	90,000
Full Linux distribution	~10 Million

2. OSS has a unique development process with unique strengths/weaknesses

The OSS process is unique in its participants' motivations and the resources that can be brought to bare down on problems. OSS, therefore, has some interesting, non-replicable assets which should be thoroughly understood.

{ TN comments:
An interesting piece of terminology -- ``non-replicable assets'' -- implies that Microsoft's **modus operandi** typically involves copying anything that others do. }

History

Open source software has roots in the hobbyist and the scientific community and was typified by ad hoc exchange of source code by developers/users.

Internet Software

The largest case study of OSS is the Internet. Most of the earliest code on the Internet was, and is still based on OSS as described in an interview with Tim O'Reilly (http://www.techweb.com/internet/profile/toreilly/interview):

TIM O'REILLY: The biggest message that we started out with was, "open source software works." ... BIND has absolutely dominant market share as the single most mission-critical piece of software on the Internet. Apache is the dominant Web server. SendMail runs probably eighty percent of the mail servers and probably touches every single piece of e-mail on the Internet

Free Software Foundation / GNU Project

Credit for the first instance of modern, organized OSS is generally given to Richard Stallman of MIT. In late 1983, Stallman created the Free Software Foundation (FSF) -- http://www.gnu.ai.mit.edu/fsf/fsf.html -- with the goal of creating a free version of the UNIX operating system. The FSF released a series of sources and binaries under the GNU moniker (which recursively stands for "Gnu's Not Unix").

The original FSF / GNU initiatives fell short of their original goal of creating a completely OSS Unix. They did, however, contribute several famous and widely disseminated applications and programming tools used today including:

- **GNU Emacs** -- originally a powerful character-mode text editor, over time Emacs was enhanced to provide a front-end to compilers, mail readers, etc.
- **GNU C Compiler (GCC)** -- GCC is the most widely used compiler in academia & the OSS world. In addition to the compiler a fairly standardized set of intermediate libraries are available as a superset to the ANSI C libraries.
- **GNU GhostScript** -- Postscript printer/viewer.

CopyLeft Licensing

FSF/GNU software introduced the "copyleft" licensing scheme that not only made it illegal to hide source code from GNU software but also made it illegal to hide the source from work *derived from GNU software.* The document that described this license is known as the General Public License (GPL).

Wired magazine has the following summary of this scheme & its intent (http://www.wired.com/wired/5.08/linux.html):

The general public license, or GPL, allows users to sell, copy, and change copylefted programs - which can also be copyrighted - but you must pass along the same freedom to sell or copy your modifications and change them further. You must also make the source code of your modifications freely available.

The second clause -- open source code of derivative works -- has been the most controversial (and, potentially the most successful) aspect of CopyLeft licensing.

Open Source Process

Commercial software development processes are hallmarked by organization around economic goals. However, since money is often not the (primary) motivation behind Open Source Software, understanding the nature of the threat posed requires a deep understanding of the process and motivation of Open Source development teams.

In other words, to understand how to compete against OSS, we must target a process rather than a company.

{ This is a very important insight, one I wish Microsoft had missed. The real battle isn't NT vs. Linux, or Microsoft vs. Red Hat/Caldera/S.u.S.E. -- it's closed-source development versus open-source. The cathedral versus the bazaar.

This applies in reverse as well, which is why bashing Microsoft qua Microsoft misses the point -- they're a symptom, not the disease itself. I wish more Linux hackers understood this.

On a practical level, this insight means we can expect Microsoft's propaganda machine to be directed against the process and culture of open source, rather than specific competitors. Brace for it... }

Open Source Development Teams

Some of the key attributes of Internet-driven OSS teams:

- Geographically far-flung. Some of the key developers of Linux, for example, are uniformly distributed across Europe, the US, and Asia.

{ It's very interesting that the author recognizes this, but doesn't go on to discuss either Linux's edge in internationalization or the extent to which Linux's success overseas (especially in Europe) is driven by a fear of U.S. technological domination. This omission may represent an exploitable blind spot in Microsoft's strategy. }

o Large set of contributors with a smaller set of core individuals. Linux, once again, has had over 1000 people submit patches, bug fixes, etc. and has had over 200 individuals directly contribute code to the kernel.

o Not monetarily motivated (in the short run). These individuals are more like hobbyists spending their free time / energy on OSS project development while maintaining other full time jobs. This has begun to change somewhat as commercial versions of the Linux OS have appeared.

OSS Development Coordination

Communication -- Internet Scale

Coordination of an OSS team is extremely dependent on Internet-native forms of collaboration. Typical methods employed run the full gamut of the Internet's collaborative technologies:

o Email lists

o Newsgroups

o 24x 7 monitoring by international subscribers

o Web sites

OSS projects the size of Linux and Apache are only viable if a large enough community of highly skilled developers can be amassed to attack a problem. Consequently, there is direct correlation between the size of the project that OSS can tackle and the growth of the Internet.

Common Direction

In addition to the communications medium, another set of factors implicitly coordinate the direction of the team.

Common Goals

Common goals are the equivalent of vision statements which permeate the distributed decision making for the entire development team. A single, clear directive (e.g. "recreate UNIX") is far more efficiently communicated and acted upon by a group than multiple, intangible ones (e.g. "make a good operating system").

Common Precedents

Precedence is potentially the most important factor in explaining the rapid and cohesive growth of massive OSS projects such as the Linux Operating System. Because the entire Linux community has years of shared experience dealing with many other forms of UNIX, they are easily able to discern -- in a non-confrontational manner -- what worked and what didn't.

There weren't arguments about the command syntax to use in the text editor -- everyone already used "vi" and the developers simply parcelled out chunks of the command namespace to develop.

Having historical, 20:20 hindsight provides a strong, implicit structure. In more forward looking organizations, this structure is provided by strong, visionary leadership.

{ At first glance, this just reads like a brown-nose-Bill comment by someone expecting that Gates will read the memo -- you can almost see the author genuflecting before an icon of the Fearless Leader.

More generally, it suggests a serious and potentially exploitable underestimation of the open-source community's ability to enable its own visionary leaders. We didn't get Emacs or Perl or the World Wide Web from ``20:20 hindsight'' -- nor is it correct to view even the relatively conservative Linux kernel design as a backward-looking recreation of past models.

Accordingly, it suggests that Microsoft's response to open source can be wrong-footed by emphasizing innovation in both our actions and the way we represent what we're doing to the rest of the world. }

Common Skillsets

NatBro points out that the need for a commonly accepted skillset as a pre-requisite for OSS development. This point is closely related to the common precedents phenomena. From his email:

A key attribute ... is the common UNIX/gnu/make skillset that OSS taps into and reinforces. I think the whole process wouldn't work if the barrier to entry were much higher than it is ... a modestly skilled UNIX programmer can grow into doing great things with Linux and many OSS products. Put another way -- it's not too hard for a developer in the OSS space to scratch their itch, because things build very similarly to one another, debug similarly, etc.

Whereas precedents identify the end goal, the common skillsets attribute describes the number of people who are versed in the process necessary to reach that end.

The Cathedral and the Bazaar

A very influential paper by an open source software advocate -- Eric Raymond -- was first published in May 1997 (http://www.redhat.com/redhat/cathedral-bazaar/). Raymond's paper was expressly cited by (then) Netscape CTO Eric Hahn as a motivation for their decision to release browser source code.

Raymond dissected his OSS project in order to derive rules-of-thumb which could be exploited by other OSS projects in the future. Some of Raymond's rules include:

Every good work of software starts by scratching a developer's personal itch

This summarizes one of the core motivations of developers in the OSS process -- solving an immediate problem at hand faced by an individual developer -- this has allowed OSS to evolve complex projects without constant feedback from a marketing / support organization.

{ TN remarks:
In other words, open-source software is driven by making great products, whereas Microsoft is driven by focus groups, psychological studies, and marketing. As if we didn't know that already.... }

Good programmers know what to write. Great ones know what to rewrite (and reuse).

Raymond posits that developers are more likely to reuse code in a rigorous open source process than in a more traditional development environment because they are always guaranteed access to the entire source all the time.

Widely available open source reduces search costs for finding a particular code snippet.

``Plan to throw one away; you will, anyhow."

Quoting Fred Brooks, ``The Mythical Man-Month", Chapter 11. Because development teams in OSS are often extremely far flung, many major subcomponents in Linux had several initial prototypes followed by the selection and refinement of a single design by Linus.

Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging.

Raymond advocates strong documentation and significant developer support for OSS projects in order to maximize their benefits.

Code documentation is cited as an area which commercial developers typically neglect which would be a fatal mistake in OSS.

Release early. Release often. And listen to your customers.

This is a classic play out of the Microsoft handbook. OSS advocates will note, however, that their release-feedback cycle is potentially an order of magnitude faster than commercial software's.

{ This is an interestingly arrogant statement, as if they think I was somehow inspired by the Microsoft way of binary-only releases.

But it suggests something else -- that even though the author intellectually grasps the importance of source code releases, he doesn't truly grok how powerful a lever the early release specifically of **source code** truly is. Perhaps living within Microsoft's assumptions makes that impossible.

TN comments:

The difference here is, in every release cycle Microsoft always listens to its **most ignorant customers**. This is the key to dumbing down each release cycle of software for further assaulting the non-PC population. Linux and OS/2 developers, OTOH, tend to listen to their **smartest** customers. This necessarily limits the initial appeal of the operating system, while enhancing its long-term benefits. Perhaps only a monopolist like Microsoft could get away with selling worse products each generation -- products focused so narrowly on the least-technical member of the consumer base that they necessarily sacrifice technical excellence. Linux and OS/2 tend to appeal to the customer who knows greatness when he or she sees it. The good that Microsoft does in bringing computers to the non-users is outdone by the curse they bring upon the experienced users, because their monopoly position tends to force everyone toward the lowest-common-denominator, not just the new users.

Note: This means that Microsoft does the ``heavy lifting" of expanding the overall PC marketplace. The great fear at Microsoft is that somebody will come behind them and make products that not only are more reliable, faster, and more secure, but are also easy to use, fun, and make people more productive. That would mean that Microsoft had merely served as a pioneer and taken all the arrows in the back, while we who have better products become a second wave to homestead on Microsoft's tamed territory. Well, sounds like a good idea to me.

So, we ought to take a page from Microsoft's book and listen to the newbies once in a while. But not so often that we lose our technological superiority over Microsoft.

ESR again. I don't agree with TN's apparent assumption that ease-of-use and technical superiority are **necessarily** mutually exclusive; with good design it's possible to do both. But given limited resources and poor-to-mediocre design skills, they do tend to get set in opposition with each other. Thus there's enough point to TN's analysis to make it worth reproducing here. }

Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone.

This is probably the heart of Raymond's insight into the OSS process. He paraphrased this rule as "debugging is parallelizable". More in depth analysis follows.

{ Well, he got **that** right, anyway. }

Parallel Development

Once a component framework has been established (e.g. key API's & structures defined), OSS projects such as Linux utilize multiple small teams of individuals independently solving particular problems.

Because the developers are typically hobbyists, the ability to `fund' multiple, competing efforts is not an issue and the OSS process benefits from the ability to pick the best potential implementation out of the many produced.

Note, that this is very dependent on:

- A large group of individuals willing to submit code
- A strong, implicit componentization framework (which, in the case of Linux was inherited from UNIX architecture).

Parallel Debugging

The core argument advanced by Eric Raymond is that unlike other aspects of software development, code debugging is an activity whose efficiency improves nearly linearly with the number of individuals tasked with the project. There are little/no management or coordination costs associated with debugging a piece of open source code -- this is the key `break' in Brooks' laws for OSS.

Raymond includes Linus Torvald's description of the Linux debugging process:

My original formulation was that every problem ``will be transparent to somebody". Linus demurred that the person who understands and fixes the problem is not necessarily or even usually the person who first characterizes it. ``Somebody finds the problem," he says, ``and somebody else understands it. And I'll go on record as saying that finding it is the bigger challenge." But the point is that both things tend to happen quickly

Put alternately:

``Debugging is parallelizable". Jeff [Dutky <dutky@wam.umd.edu>] observes that although debugging requires debuggers to communicate with some coordinating developer, it doesn't require significant coordination between debuggers. Thus it doesn't fall prey to the same quadratic complexity and management costs that make adding developers problematic.

One advantage of parallel debugging is that bugs and their fixes are found / propagated much faster than in traditional processes. For example, when the TearDrop IP attack was first posted to the web, less than 24 hours passed before the Linux community had a working fix available for download.

"Impulse Debugging"

An extension to parallel debugging that I'll add to Raymond's hypothesis is "impulsive debugging". In the case of the Linux OS, implicit to the act of installing the OS is the act of installing *the debugging/development environment.* Consequently, it's highly likely that if a particular user/developer comes across a bug in another individual's component – and especially if that bug is "shallow" -- that user can very quickly patch the code and, via internet collaboration technologies, propagate that patch very quickly back to the code maintainer.

Put another way, OSS processes have a very low entry barrier to the debugging process due to the common development/debugging methodology derived from the GNU tools.

Conflict resolution

Any large scale development process will encounter conflicts which must be resolved. Often resolution is an arbitrary decision in order to further progress the project. In commercial teams, the corporate hierarchy + performance review structure solves this problem -- How do OSS teams resolve them?

In the case of Linux, Linus Torvalds is the undisputed `leader' of the project. He's delegated large components (e.g. networking, device drivers, etc.) to several of his trusted "lieutenants" who further de-facto delegate to a handful of "area" owners (e.g. LAN drivers).

Other organizations are described by Eric Raymond: (http://earthspace.net/~esr/writings/homesteading/homesteading-15.html):

Some very large projects discard the `benevolent dictator' model entirely. One way to do this is turn the co-developers into a voting committee (as with Apache). Another is rotating dictatorship, in which control is occasionally passed from one member to another within a circle of senior co-developers (the Perl developers organize themselves this way).

Motivation

This section provides an overview of some of the key reasons OSS developers seek to contribute to OSS projects.

Solving the Problem at Hand

This is basically a rephrasing of Raymond's first rule of thumb -- "Every good work of software starts by scratching a developer's personal itch".

Many OSS projects -- such as Apache -- started as a small team of developers setting out to solve an immediate problem at hand. Subsequent improvements of the code often stem from individuals applying the code to their own scenarios (e.g. discovering that there is no device driver for a particular NIC, etc.)

Education

The Linux kernel grew out of an educational project at the University of Helsinki. Similarly, many of the components of Linux / GNU system (X windows GUI, shell utilities, clustering, networking, etc.) were extended by individuals at educational institutions.

- In the Far East, for example, Linux is reportedly growing faster than internet connectivity – due primarily to educational adoption.
- Universities are some of the original proponents of OSS as a teaching tool.
- Research/teaching projects on top of Linux are easily `disseminated' due to the wide availability of Linux source. In particular, this often means that new research ideas are first implemented and available on Linux before they are available / incorporated into other platforms.

{ This from the same author who later insists that the Linux mob will have a hard time absorbing new ideas!. }

Ego Gratification

The most ethereal, and perhaps most profound motivation presented by the OSS development community is pure ego gratification.

In "The Cathedral and the Bazaar", Eric S. Raymond cites:

The ``utility function" Linux hackers are maximizing is not classically economic, but is the intangible of their own ego satisfaction and reputation among other hackers.

And, of course, "you aren't a hacker until someone else calls you hacker"

Homesteading on the Noosphere

A second paper published by Raymond -- "Homesteading on the Noosphere" (http://sagan.earthspace.net/~esr/writings/homesteading/), discusses the difference between economically motivated exchange (e.g. commercial software development for money) and "gift exchange" (e.g. OSS for glory).

"Homesteading" is acquiring property by being the first to `discover' it or by being the most recent to make a significant contribution to it. The "Noosphere" is loosely defined as the "space of all work". Therefore, Raymond posits, the OSS hacker motivation is to lay a claim to the largest area in the body of work. In other words, take credit for the biggest piece of the prize.

{ This is a subtle but significant misreading. It introduces a notion of territorial `size' which is nowhere in my theory. It may be a personal error of the author, but I suspect it reflects Microsoft's competition-obsessed culture. }

From "Homesteading on the Noosphere":

Abundance makes command relationships difficult to sustain and exchange relationships an almost pointless game. In gift cultures, social status is determined not by what you control but by **what you give away**.

...

For examined in this way, it is quite clear that the society of open-source hackers is in fact a gift culture. Within it, there is no serious shortage of the `survival necessities' – disk space, network bandwidth, computing power. Software is freely shared. This abundance creates a situation in which the only available measure of competitive success is reputation among one's peers.

More succinctly (http://www.techweb.com/internet/profile/eraymond/interview):

SIMS: So the scarcity that you looked for was the scarcity of attention and reward?
RAYMOND: That's exactly correct.

Altruism

This is a controversial motivation and I'm inclined to believe that at some level, Altruism `degenerates' into a form of the Ego Gratification argument advanced by Raymond.

One smaller motivation which, in part, stems from altruism is Microsoft-bashing.

{ What a very fascinating admission, coming from a Microserf! Of course, he doesn't analyze why this connection exists; that might hit too close to home...}

Code Forking

A key threat in any large development team -- and one that is particularly exacerbated by the process chaos of an internet-scale development team -- is the risk of code-forking.

Code forking occurs when over normal push-and-pull of a development project, multiple, inconsistent versions of the project's code base evolve.

In the commercial world, for example, the strong, singular management of the Windows NT codebase is considered to be one of it's greatest advantages over the `forked' codebase found in commercial UNIX implementations (SCO, Solaris, IRIX, HP-UX, etc.).

Forking in OSS -- BSD Unix

Within OSS space, BSD Unix is the best example of forked code. The original BSD UNIX was an attempt by U-Cal Berkeley to create a royalty-free version of the UNIX operating system for teaching purposes. However, Berkeley put severe restrictions on non-academic uses of the codebase.

{ The author's history of the BSD splits is all wrong. }

In order to create a fully free version of BSD UNIX, an ad hoc (but closed) team of developers created FreeBSD. Other developers at odds with the FreeBSD team for one reason or another splintered the OS to create other variations (OpenBSD, NetBSD, BSDI).

There are two dominant factors which led to the forking of the BSD tree:

o Not everyone can contribute to the BSD codebase. This limits the size of the effective "Noosphere" and creates the potential for someone else to credibly claim that their forked code will become more dominant than the core BSD code.

{ Wow. This is an insight I never had -- that forking can actually be driven by the belief that the forker could accumulate a bigger bazaar than the current project. It certainly explains EGCS and the BSD-spinoff-group-of-the-week phenomenon, though probably not the Emacs/XEmacs split.

OK, we've learned something now. This may in fact explain the couinterintuitive fact that the projects which open up development the most actually have the **least** tendency to fork... }

o Unlike GPL, BSD's license places no restrictions on derivative code. Therefore, if you think your modifications are cool enough, you are free to fork the code, charge money for it, change its name, etc.

Both of these motivations create a situation where developers may try to force a fork in the code and collect royalties (monetary, or ego) at the expense of the collective BSD society.

(Lack of) Forking in Linux

In contrast to the BSD example, the Linux kernel code base hasn't forked. Some of the reasons why the integrity of the Linux codebase has been maintained include:

o Universally accepted leadership

Linus Torvalds is a celebrity in the Linux world and his decisions are considered final. By contrast, a similar celebrity leader did NOT exist for the BSD-derived efforts.

Linus is considered by the development team to be a fair, well-reasoned code manager and his reputation within the Linux community is quite strong. However, Linus doesn't get involved in every decision. Often, sub groups resolve their -- often large -- differences amongst themselves and prevent code forking.

o Open membership & long term contribution potential.

In contrast to BSD's closed membership, anyone can contribute to Linux and your "status" -- and therefore ability to `homestead' a bigger piece of Linux -- is based on the size of your previous contributions.

Indirectly this presents a further disincentive to code forking. There is almost no credible mechanism by which the forked, minority code base will be able to maintain the rate of innovation of the primary Linux codebase.

o GPL licensing eliminates economic motivations for code forking

Because derivatives of Linux MUST be available through some free avenue, it lowers the long term economic gain for a minority party with a forked Linux tree.

o Forking the codebase also forks the "Noosphere"

Ego motivations push OSS developers to plant the biggest stake in the biggest Noosphere. Forking the code base inevitably shrinks the space of accomplishment for any subsequent developers to the new code tree.

Open Source Strengths

What are the core strengths of OSS products that Microsoft needs to be concerned with?

OSS Exponential Attributes

Like our Operating System business, OSS ecosystems have several exponential attributes:

- **OSS processes are growing with the Internet**

The single biggest constraint faced by any OSS project is finding enough developers interested in contributing their time towards the project. As an enabler, the Internet was absolutely necessary to bring together enough people for an Operating System scale project. More importantly, the *growth engine* for these projects is the growth in the Internet's reach. Improvements in collaboration technologies directly lubricate the OSS engine.

Put another way, the growth of the Internet will make existing OSS projects bigger and will make OSS projects in "smaller" software categories become viable.

- **OSS processes are "winner-take-all"**

Like commercial software, the most viable single OSS project in many categories will, in the long run, kill competitive OSS projects and `acquire' their IQ assets. For example, Linux is killing BSD Unix and has absorbed most of its core ideas (as well as ideas in the commercial UNIXes). This feature confers huge first mover advantages to a particular project

- **Developers seek to contribute to the largest OSS platform**

The larger the OSS project, the greater the prestige associated with contributing a large, high quality component to its Noosphere. This phenomena contributes back to the "winner-take-all" nature of the OSS process in a given segment.

- **Larger OSS projects solve more "problems at hand"**

The larger the project, the more development/test/debugging the code receives. The more debugging, the more people who deploy it.

Long-term credibility

Binaries may die but source code lives forever

One of the most interesting implications of viable OSS ecosystems is long-term credibility.

Long-Term Credibility Defined

Long term credibility exists if there is no way you can be driven out of business in the near term. This forces change in how competitors deal with you.

{ TN comments:

Note the terminology used here ``driven out of business''. MS believes that putting other companies out of business is not merely ``collateral damage'' -- a byproduct of selling better stuff -- but rather, a direct business goal. To put this in perspective, economic theory and the typical honest, customer-oriented businessperson will think of business as a stock-car race -- the fastest car with the most skillful driver wins. Microsoft views business as a demolition derby -- you knock out as many competitors as possible, and try to maneuver things so that your competitors wipe each other out and thereby eliminate themselves. In a stock car race there are many finishers and thus many drivers get a paycheck. In a demolition derby there is just one survivor. Can you see why ``Microsoft'' and ``freedom of choice'' are absolutely in two different universes? }

For example, Airbus Industries garnered initial long term credibility from explicit government support. Consequently, when bidding for an airline contract, Boeing would be more likely to accept short-term, non-economic returns when bidding against Lockheed than when bidding against Airbus.

Loosely applied to the vernacular of the software industry, a product/process is long-term credible if FUD tactics can not be used to combat it.

OSS is Long-Term Credible

OSS systems are considered credible because the source code is available from potentially millions of places and individuals.

{ We are deep inside the Microsoft world-view here. I realize that a typical hacker's reaction to this kind of thinking will be to find it nauseating, but it reflects a kind of instrumental ruthlessness about the uses of negative marketing that we need to learn to cope with.

The **really** interesting thing about these two statements is that they imply that Microsoft should give up on FUD as an effective tactic against us.

Most of us have been assuming that the DOJ antitrust suit is what's keeping Microsoft from hauling out the FUD guns. But if His Gatesness bought this part of the memo, Microsoft may believe that they need to develop a more substantive response because FUD won't work.

This could be both good and bad news. The good news is that Microsoft would give up attack marketing, a weapon which in the past has been much more powerful than its distinctly inferior technology. The bad news is that, against **us**, giving it up would actually be better strategy; they wouldn't be wasting energy any more and might actually evolve some effective response. }

The likelihood that Apache will cease to exist is orders of magnitudes lower than the likelihood that WordPerfect, for example, will disappear. The disappearance of Apache is not tied to the disappearance of binaries (which are affected by purchasing shifts, etc.) but rather to the disappearance of source code and the knowledge base.

Inversely stated, customers know that Apache will be around 5 years from now -- provided there exists some minimal sustained interested from its user/development community.

One Apache customer, in discussing his rationale for running his e-commerce site on OSS stated, "because it's open source, I can assign one or two developers to it and maintain it myself indefinitely. "

Lack of Code-Forking Compounds Long-Term Credibility

The GPL and its aversion to code forking reassures customers that they aren't riding an evolutionary `dead-end' by subscribing to a particular commercial version of Linux.

The "evolutionary dead-end" is the core of the software FUD argument.

{ Very true -- and there's another glaring omission here. If the author had been really honest, he'd have noted that OSS advocates are well positioned to turn this argument around and beat Microsoft to death with it.

By the author's own admission, OSS is bulletproof on this score. On the other hand, the exploding complexity and schedule slippage of the just-renamed ``Windows 2000" suggest that **it** is an evolutionary dead end.

The author didn't go on to point that out. But **we** should. }

Parallel Debugging

Linux and other OSS advocates are making a progressively more credible argument that OSS software is at least as robust -- if not more -- than commercial alternatives. The Internet provides an ideal, high-visibility showcase for the OSS world.

{ It's a handful of amateurs, most of us unpaid and almost all part-time, against an entrenched multimillion-dollar propaganda machine run by some of the top specialists in the technology-marketing business.

And the amateurs are ``making a progressively more credible argument''. By Microsoft's own admission, we're actually **winning**.

Maybe there's a message about the underlying products here? }

In particular, larger, more savvy, organizations who rely on OSS for business operations (e.g. ISPs) are comforted by the fact that they can potentially fix a work-stopping bug independent of a commercial provider's schedule (for example, UUNET was able to obtain, compile, and apply the teardrop attack patch to their deployed Linux boxes within 24 hours of the first public attack)

Parallel Development

Alternatively stated, "developer resources are essentially free in OSS". Because the pool of potential developers is massive, it is economically viable to simultaneously investigate multiple solutions / versions to a problem and chose the best solution in the end.

For example, the Linux TCP/IP stack was probably rewritten 3 times. Assembly code components in particular have been continuously hand tuned and refined.

OSS = `perfect' API evangelization / documentation

OSS's API evangelization / developer education is basically providing the developer with the underlying code. Whereas evangelization of API's in a closed source model basically defaults to trust, OSS API evangelization lets the developer make up his own mind.

NatBro and Ckindel point out a split in developer capabilities here. Whereas the "enthusiast developer" is comforted by OSS evangelization, novice/intermediate developers –the bulk of the development community – prefer the trust model + organizational credibility (e.g. "Microsoft says API X looks this way")

{ Whether it's really true that most developers prefer the `trust' model or not is an extremely interesting question.

Twenty years of experience in the field tells me not; that, in general, developers prefer code even when their non-technical **bosses** are naive enough to prefer `trust'. Microsoft, obviously, wants to believe that its `organizational credibility' counts -- I detect some wishful thinking here.

On the other hand, they may be right. We in the open-source community can't afford to dismiss that possibility. I think we can meet it by developing high-quality documentation. In this way, `trust' in name authors (or in publishers of good repute such as O'Reilly or Addison-Wesley) can substitute for `trust' in an API-defining organization. }

Release rate

Strongly componentized OSS projects are able to release subcomponents as soon as the developer has finished his code. Consequently, OSS projects rev quickly & frequently.

Open Source Weaknesses

The weaknesses in OSS projects fall into 3 primary buckets:

- Management costs
- Process Issues
- Organizational Credibility

Management Costs

The biggest roadblock for OSS projects is dealing with exponential growth of management costs as a project is scaled up in terms of rate of innovation and size. This implies a limit to the rate at which an OSS project can innovate.

Starting an OSS project is difficult

From Eric Raymond:

It's fairly clear that one cannot code from the ground up in bazaar style. One can test, debug and improve in bazaar style, but it would be very hard to **originate** a project in bazaar mode. Linus didn't try it. I didn't either. Your nascent developer community needs to have something runnable and testable to play with.

Raymond `s argument can be extended to the difficulty in starting/sustaining a project if there are no clear precedent / goal (or too many goals) for the project.

Bazaar Credibility

Obviously, there are far more fragments of source code on the Internet than there are OSS communities. What separates "dead source code" from a thriving bazaar?

One article (http://www.mibsoftware.com/bazdev/0003.htm) provides the following *credibility* criteria:

"....thinking in terms of a hard minimum number of participants is misleading. Fetchmail and Linux have huge numbers of beta testers *now*, but they obviously both had very few at the beginning.

What both projects did have was a handful of enthusiasts and a plausible promise. The promise was partly technical (this code will be wonderful with a little effort) and sociological (if you join our gang, you'll have as much fun as we're having). So what's necessary for a bazaar to develop is that it be credible that the full-blown bazaar will exist!"

I'll posit that some of the key criteria that must exist for a bazaar to be credible include:

o **Large Future Noosphere** – The project must be cool enough that the intellectual reward adequately compensates for the time invested by developers. The Linux OS excels in this respect.

o **Scratch a big itch** -- The project must be important / deployable by a large audience of *developers*. The Apache web server provides an excellent example here.

o **Solve the right amount of the problem first** -- Solving too much of the problem relegates the OSS development community to the role of testers. Solving too little before going OSS reduces "plausible promise" and doesn't provide a strong enough component framework to efficiently coordinate work.

{ These three points are well-thought-out and actually improve on my characterization in ``The Cathedral and the Bazaar.''. The distinction he makes between `Large Future Noosphere' and `Scratch a big itch' is particularly telling. }

Post-Parity Development

When describing this problem to JimAll, he provided the perfect analogy of "chasing tail lights". The easiest way to get coordinated behavior from a large, semi-organized mob is to point them at a known target. Having the taillights provides concreteness to a fuzzy vision. In such situations, having a taillight to follow is a proxy for having strong central leadership.

Of course, once this implicit organizing principle is no longer available (once a project has achieved "parity" with the state-of-the-art), the level of management necessary to push towards new frontiers becomes massive.

{ Nonsense. In the open-source world, all it takes is one person with a good idea. Part of the point of open source is to lower the energy barriers that retard innovation. We've found by experience that the `massive management' the author extols is one of the worst of these barriers.

In the open-source world, innovators get to try anything, and the only test is whether users will volunteer to experiment with the innovation and like it once they have. The Internet facilitates this process, and the cooperative conventions of the open-source community are specifically designed to promote it.

The third alternative to ``chasing taillights'' or ``strong central leadership'' (and more effective than either) is an evolving creative anarchy, in which there are a thousand leaders and ten thousand followers linked by a web of peer review and subject to rapid-fire reality checks.

Microsoft cannot beat this. I don't think they can even really **understand** it, not on a gut level. }

This is possibly the single most interesting hurdle to face the Linux community now that they've achieved parity with the state of the art in UNIX in many respects.

{ The Linux community has not merely lept this hurdle, but utterly **demolished** it. This fact is at the core of open-source's long-term advantage over closed-source development. }

Un-sexy work

Another interesting thing to observe in the near future of OSS is how well the team is able to tackle the "unsexy" work necessary to bring a commercial grade product to life.

{ Characterizing this kind of work as ``unsexy'' reveals an interesting blind spot. It has been my experience that for almost any kind of work, there will be somebody, somewhere, who thinks it's interesting or fulfilling enough to undertake it.

Take the example of Unicode support above. Who's likely to do the best, most thorough job of implementing Unicode support, of the following three people?

- Joe M. Serf's boss assigns WUS (Windows Unicode Support) to him.
- Ana Ng lives in Malaysia and really needs good multiple-language support in order to be able to view information in a variety of Asian languages.
- Jeff P. Hacker lives in Indiana and is fascinated by the problem of providing robust support for multiple alphabets.

It's likely to be either Ana or Jeff (all else, including skill sets, being equal), because they're scratching their itches. It ain't gonna be Joe.

Now, which development model is more likely to pull Ana or Jeff into the development effort – closed source, or open?

Easy question. }

In the operating systems space, this includes small, essential functions such as power management, suspend/resume, management infrastructure, UI niceties, deep Unicode support, etc.

For Apache, this may mean novice-administrator functionality such as wizards.

Integrative/Architectural work

Integrative work across modules is the biggest cost encountered by OSS teams. An email memo from Nathan Myrhvold on 5/98, points out that of all the aspects of software development, integration work is most subject to Brooks' laws.

Up till now, Linux has greatly benefited from the integration / componentization model pushed by previous UNIX's. Additionally, the organization of Apache was simplified by the relatively simple, fault tolerant specifications of the HTTP protocol and UNIX server application design.

Future innovations which require changes to the core architecture / integration model are going to be incredibly hard for the OSS team to absorb because it simultaneously devalues their precedents and skillsets.

{ This prediction is of a piece with the author's earlier assertion that open-source development relies critically on design precedents and is unavoidably backward-looking. It's myopic -- apparently things like Python, Beowulf, and Squeak (to name just three of hundreds of innovative projects) don't show on his radar.

We can only hope Microsoft continues to believe this, because it would hinder their response. Much will depend on how they interpret innovations such as (for example) the SMPization of the Linux kernel.

Interestingly, the author contradicts himself on this point. A former Microserf tells me that `throw one away' is actually pretty close to a defined Microsoft policy, but one designed to leverage marketing rather than fix problems. The project he was involved with involved a web-based front-end to Exchange. The resulting first draft (after 14 months of effort) was completely inferior to already existing free-web-email (Yahoo, Hotmail, etc). The official response to that was `` That's ok. We'll get the market share and fix the technical problems over the next 3-4 years".

He adds: Internet Explorer 5, just before one of its beta releases had about 300K (yes, 300K) outstanding bugs targeted to be fixed before the beta release. Much of this was accomplished by simply removing large chunks of planned (new) functionality and pushing them to a later (+1-2 years later) release. }

Process Issues

These are weaknesses intrinsic to OSS's design/feedback methodology.

Iterative Cost

One of the key's to the OSS process is having many more iterations than commercial software (Linux was known to rev it's kernel more than once a day!). However, commercial customers tell us they want fewer revs, not more.

{ I wonder how this answer would change if Microsoft revs weren't so expensive?

This is why commercial Linux distributors exist – to mediate between the rapid-development process and customers who don't want to follow every twist of it. The kernel may rev once a day, but Red Hat only revs once in six months. }

"Non-expert" Feedback

The Linux OS is not developed for end users but rather, for other hackers. Similarly, the Apache web server is implicitly targetted at the largest, most savvy site operators, not the departmental intranet server.

The key thread here is that because OSS doesn't have an explicit marketing / customer feedback component, wishlists -- and consequently feature development -- are dominated by the most technically savvy users.

One thing that development groups at MSFT have learned time and time again is that ease of use, UI intuitiveness, etc. must be built from the ground up into a product and can not be pasted on at a later time.

{ This demands comment -- because it's so right in theory, but so hideously wrong in Microsoft practice. The wrongness implies an exploitable weakness in the implied strategy (for Microsoft) of emphasizing UI.

There are two ways to build in ease of use "from the ground up". One (the Microsoft way) is to design monolithic applications that are defined and dominated by their UIs. This tends to produce ``Windowsitis'' -- rigid, clunky, bug-prone monstrosities that are all glossy surface with a hollow interior.

Programs built this way **look** user-friendly at first sight, but turn out to be huge time and energy sinks in the longer term. They can only be sustained by carpet-bomb marketing, the main purpose of which is to delude users into believing that (a) bugs are features, or that (b) all bugs are really the stupid user's fault, or that (c) all bugs will be abolished if the user bends over for the next upgrade. This approach is fundamentally broken.

The other way is the Unix/Internet/Web way, which is to separate the engine (which does the work) from the UI (which does the viewing and control). This approach requires that the engine and UI communicate using a well-defined protocol. It's exemplified by browser/server pairs -- the engine specializes in being an engine, and the UI specializes in being a UI.

With this second approach, overall complexity goes down and reliability goes up. Further, the interface is easier to evolve/improve/customize, precisely because it's not tightly coupled to the engine. It's even possible to have multiple interfaces tuned to different audiences.

Finally, this architecture leads naturally to applications that are enterprise-ready -- that can be used or administered remotely from the server. This approach works – and it's the open-source community's natural way to counter Microsoft.

The key point is here is that if Microsoft wants to fight the open-source community on UI, let them – because we can win that battle, too, fighting it our way. They can write ever-more-elaborate Windows monoliths that spot-weld you to your application-server console. We'll win if we write clean distributed applications that leverage the Internet and the Web and make the UI a pluggable/unpluggable user choice that can evolve.

Note, however, that our win depends on the existence of well-defined protocols (such as HTTP) to communicate between UIs and engines. That's why the stuff later in this memo about ``de-commoditizing protocols'' is so sinister. We need to guard against that. }

The interesting trend to observe here will be the effect that commercial OSS providers (such as RedHat in Linux space, C2Net in Apache space) will have on the feedback cycle.

Organizational Credibility

How can OSS provide the *service* that consumers expect from software providers?

Support Model

Product support is typically the first issue prospective consumers of OSS packages worry about and is the primary feature that commercial redistributors tout.

However, the vast majority of OSS projects are supported by the developers of the respective components. Scaling this support infrastructure to the level expected in commercial products will be a significant challenge. There are many orders of magnitude difference between users and developers in IIS vs. Apache.

{ The vagueness of this last sentence is telling. Had the author continued, he would have had to acknowledge that Apache is clobbering the crap out of IIS in the marketplace (Apache's share 54% and climbing; IIS's somewhere around 14% and dropping).

This would have led to a choice of unpalatable (for Microsoft) alternatives. It may be that Apache's informal user-support channels and `organizational credibility' actually produce better results than Microsoft's IIS organization can offer. If that's true, then it's hard to see in principle why the same shouldn't be true of other open-source projects.

The alternative -- that Apache is so good that it doesn't **need** much support or `organizational credibility' -- is even worse. That would mean that all of Microsoft's heavy-duty support and marketing battalions were just a huge malinvestment, like crumbling Stalinist apartment blocks forty years later.

These two possible explanations imply distinct but parallel strategies for open-source advocates. One is to build software that's so good it just doesn't need much support (but we'd do this anyway, and generally have). The other is to do more intensely what we're already doing along the lines of support mailing lists, newsgroups, FAQs, and other informal but extremely effective channels. A former Microserf adds: As of NT5 (sorry, Win2K :-) MS is going to claim a huge increase in IIS market share. This is because IIS5 is built directly linked with the NT kernel and handles all external TCP traffic (mail, http, etc). MSOffice is also going to communicate through IIS when talking with NT or Exchange, thus allowing them to add all internal LAN traffic to their usage reports. Let's see if we can pop their balloon before they raise it. }

For the short-medium run, this factor alone will relegate OSS products to the top tiers of the user community.

Strategic Futures

A very sublime problem which will affect full scale consumer adoption of OSS projects is the lack of strategic direction in the OSS development cycle. While incremental improvement of the current bag of features in an OSS product is very credible, *future features* have no organizational commitment to guarantee their development.

{ No. In the open-source community, new features are driven by the novelty- and territory-seeking behavior of individual hackers. This certainly is not a force to be despised. The Internet and the Web were built this way -- not because of `organizational commitment', but because somebody, somewhere, thought ``Hey – this would be neat...".

Perhaps we're fortunate that `organizational credibility' looms so large in the Microsoft world-view. The time and energy they spend worrying about that and believing it's a prerequisite is resources they won't spend doing anything that might be effective against us. }

What does it mean for the Linux community to "sign up" to help build the Corporate Digital Nervous System? How can Linux guarantee backward compatibility with apps written to previous API's? Who do you sue if the next version of Linux breaks some commitment? How does Linux make a strategic alliance with some other entity?

{ Who do you sue if NT 5.0 (excuse me, "Windows 2000") doesn't ship on time? Has **anyone** ever recovered from Microsoft for any of their backwards-incompatibilities or other screwups?

The question about backward compatibility is pretty ironic, considering that I've never heard of a program that will run under all of Windows 3.1, Windows 95, Windows 98, and NT 4.0 without change.

The author has been overtaken by events here. He should ask Microsoft's buddies at Intel, who bought a minority stake in Red Hat less than two months after this memo was written. }

Open Source Business Models

In the last 2 years, OSS has taken another twist with the emergence of companies that sell OSS software, and more importantly, hiring full-time developers to improve the code base. What's the business model that justifies these salaries?

In many cases, the answers to these questions are similar to "why should I submit my protocol/app/API to a standards body?"

Secondary Services

The vendor of OSS-ware provides sales, support, and integration to the customer. Effectively, this transforms the OSS-ware vendor from a package goods manufacturer into a services provider.

Loss Leader -- Market Entry

The Loss Leader OSS business model can be used for two purposes:

- Jumpstarting an infant market
- Breaking into an existing market with entrenched, closed-source players

Many OSS startups -- particularly those in Operating Systems space -- view funding the development of OSS products as a strategic loss leader against Microsoft.

Linux distributors, such as RedHat, Caldera, and others, are expressly willing to fund full time developers who release all their work to the OSS community. By simultaneously funding these efforts, Red Hat and Caldera are implicitly colluding and believe they'll make more short term revenue by growing the Linux market rather than directly competing with each other.

An indirect example is O'Reilly & Associates employment of Larry Wall -- "leader" and full time developer of PERL. The #1 publisher of PERL reference books, of course is O'Reilly & Associates.

For the short run, especially as the OSS project is at the steepest part of it's growth curve, such investments generate positive ROI. Longer term, ROI motivations may steer these developers towards making proprietary extensions rather than releasing OSS.

Commoditizing Downstream Suppliers

This is very closely related to the loss leader business model. However, instead of trying to get marginal service returns by massively growing the market, these businesses increase returns in their part of the value chain by commoditizing downstream suppliers.

The best examples of this currently are the thin server vendors such as Whistle Communications, and Cobalt Micro who are actively funding developers in SAMBA and Linux respectively.

Both Whistle and Cobalt generate their revenue on hardware volume. Consequently, funding OSS enables them to avoid today's PC market where a "tax" must be paid to the OS vendor (NT Server retail price is $800 whereas Cobalt's target MSRP is around $1000).

The earliest Apache developers were employed by cash-strapped ISPs and ICPs.

Another, more recent example is IBMs deal with Apache. By declaring the HTTP server a commodity, IBM hopes to concentrate returns in the more technically arcane application services it bundles with it's Apache distribution (as well as hope to reach Apache's tremendous market share).

First Mover -- Build Now, $$ Later

One of the exponential qualities of OSS -- successful OSS projects swallow less successful ones in their space -- implies a pre-emption business model where by investing directly in OSS today, they can pre-empt / eliminate competitive projects later – especially if the project requires API evangelization. This is tantamount to seizing a first mover advantage in OSS.

In addition, the developer scale, iteration rate, and reliability advantages of the OSS process are a blessing to small startups who typically can't afford a large in--house development staff.

Examples of startups in this space include SendMail.com (making a commercially supported version of the sendmail mail transfer agent) and C2Net (makes commercial and encrypted Apache)

Notice, that no case of a successful startup *originating* an OSS project has been observed. In both of these cases, the OSS project existed *before* the startup was formed.

{ There are at least two counterexamples to this: AbiWord and Ghostscript. }

Sun Microsystem's has recently announced that its "JINI" project will be provided via a form of OSS and may represent an application of the pre-emption doctrine.

Linux

The next several sections analyze the most prominent OSS projects including Linux, Apache, and now, Netscape's OSS browser.

A second memo titled "Linux OS Competitive Analysis" provides an in-depth review of the Linux OS. Here, I provide a top-level summary of my findings in Linux.

What is it?

Linux (pronounced "LYNN-ucks") is the #1 market share Open Source OS on the Internet. Linux is derives strongly from the 25+ years of lessons learned on the UNIX operating system.

Top-Level Features:

- Multi-user / Multi-threaded (kernel & user)
- Multi-platform (x86, Alpha, MIPS, PowerPC, SPARC, etc.)
- Protected 32-bit memory space for apps; Virtual Memory support (64-bit in development)
- SMP (Intel & Sun CPU's)
- Supports multiple file systems (FAT16, FAT32, NTFS, Ext2FS)
- High performance networking
  - NFS/SMB/IPX/Appletalk networking
  - Fastest stack in Unix vs. Unix perf tests
- Disk Management
  - Striping, mirroring, FAT16, FAT32, NTFS
- Xfree86 GUI

Linux is a real, credible OS + Development process

Like other Open Source Software (OSS) products, the real key to Linux isn't the static version of the product but rather the process around it. This process lends credibility and an air of future-safeness to customer Linux investments.

- **Trusted in mission criticial environments**. Linux has been deployed in mission critical, commercial environments with an excellent pool of public testimonials.

o **Linux = Best of Breed UNIX.** Linux outperforms many other UNIX's in most major performance category (networking, disk I/O, process ctx switch, etc.). To grow their featurebase, Linux has also liberally stolen features of other UNIX's (shell features, file systems, graphics, CPU ports)

o **Only Unix OS to gain market share.** Linux is on track to eventually own the x86 UNIX market and has been the only UNIX version to gain net Server OS market share in recent years. I believe that Linux -- moreso than NT -- will be the biggest threat to SCO in the near future.

o **Linux's process iterates VERY fast.** For example, the Linux equivalent of the TransmitFile() API went from idea to final implementation in about 2 weeks time.

{ All true. I couldn't have put it better myself :-). }

Linux is a short/medium-term threat in servers

The primary threat Microsoft faces from Linux is against NT Server.

Linux's future strength against NT server (and other UNIXes) is fed by several key factors:

o Linux uses commodity PC hardware and, due to OS modularity, can be run on smaller systems than NT. Linux is frequently used for services such as DNS running on old 486's in back closets.

o Due to it's UNIX heritage, Linux represents a lower switching cost for some organizations than NT

o UNIX's perceived Scaleability, Interopability, Availability, and Manageability (SIAM) advantages over NT.

o Linux can win as long as services / protocols are commodities

{ We sense a theme developing here...

To put it slightly differently: Linux can win if services are open and protocols are simple, transparent. Microsoft can only win if services are closed and protocols are complex, opaque.

To put it even more bluntly: "commodity" services and protocols are good things for customers; they promote competition and choice. **Therefore, for Microsoft to win, the customer must lose.**

The most interesting revelation in this memo is how close to explicitly stating this logic Microsoft is willing to come. }

Linux is unlikely to be a threat on the desktop

Linux is unlikely to be a threat in the medium-long term on the desktop for several reasons:

o **Poor end-user apps & focus.** OSS development process are far better at solving individual component issues than they are at solving integrative scenarios such as end-to-end ease of use.

{ The easy and obvious counter to this is to observe that Microsoft is pretty bad at `end-to-end ease of use' itself; what it's good at is creating systems that **look at first sight** as though they have that quality, but don't actually deliver on it (and, over time, have a far higher total cost in productivity lost to bugs and missing features than does Linux).

Though this is true, it evades an important issue -- which is that Microsoft's own meretriciousness on this score doesn't make its criticism any less valid. Open-source development really is poor at addressing this class of issues, because it doesn't involve systematic ease-of-use-testing with non-hackers.

This genuinely will slow down Linux's advance on the desktop. It is not likely to stall it forever, however -- not if efforts like GNOME and KDE get time to mature. }

o **Switching costs for desktop installed base.** Switching desktops is hard and a challenger must be able to prove a significant marginal advantage. Linux's process is more focused on second-mover advantages (e.g. copying what's been proven to work) and is therefore unlikely to provide the first-mover advantage necessary to provide switching impetus.

{ There's a hidden presumption here that innovation and ``first mover advantage'' are the only ways to defray the perceived cost of switching. This is a dangerous assumption for Microsoft; it may be that the superior reliability and stability of Linux is sufficient.

Even granting the author's presumption, the possibility that Linux can grab a sufficient `first-mover' advantage is not safely foreclosed unless the open-source mode really is incapable of generating innovation – and we already know that's not true. }

o **UNIX heritage will slow encroachment.** Ease of use must be engineered from the ground up. Linux's hacker orientation will never provide the ease-of-use requirements of the average desktop user.

{ My previous comments on ease-of-use engineering, and the open-source community's way to beat this rap, apply here. We need to wrong-foot Microsoft by building systems that use openness to support users readily **evolving** their environments to optimum, in the way that the Web does. }

Beating Linux

In addition to the attacking the general weaknesses of OSS projects (e.g. Integrative / Architectural costs), some specific attacks on Linux are:

o Beat UNIX

All the standard product issues for NT vs. Sun apply to Linux.

o Fold extended functionality into commodity protocols / services and create new protocols

Linux's homebase is currently commodity network and server infrastructure. By folding extended functionality (e.g. Storage+ in file systems, DAV/POD for networking) into today's commodity services, we raise the bar & change the rules of the game.

{ Here, as in the earlier comment on how Linux can win, we start to see the actual outlines of a Microsoft strategy emerge from the fog of corporatese. And it ain't pretty; in

fact, it's ugly enough to make it appropriate that it's pushing midnight on Halloween as I write.

What the author is driving at is nothing less than trying to subvert the entire "commodity network and server" infrastructure (featuring TCP/IP, SMTP, HTTP, POP3, IMAP, NFS, and other open standards) into using protocols which, though they might have the same names, have actually been subverted into customer- and market-control devices for Microsoft (this is what the author really means when he exhorts Microserfs to ``raise the bar & change the rules of the game").

The `folding extended functionality' here is a euphemism for introducing nonstandard extensions (or entire alternative protocols) which are then saturation-marketed as standards, even though they're closed, undocumented or just specified enough to create an illusion of openness. The objective is to make the new protocols a checklist item for gullible corporate buyers, while simultaneously making the writing of third-party symbiotes for Microsoft programs next to impossible. (And anyone who succeeds gets bought out.)

This game is called ``embrace and extend''. We've seen Microsoft play this game before, and they're very good at it. When it works, Microsoft wins a monopoly lock. Customers lose.

(This standards-pollution strategy is perfectly in line with Microsoft's efforts to corrupt Java and break the Java brand.)

Open-source advocates can counter by pointing out exactly how and why customers lose (reduced competition, higher costs, lower reliability, lost opportunities). Open-source advocates can also make this case by showing the contrapositive -- that is, how open source and open standards increase vendor competition, decrease costs, improve reliability, and create opportunities.

Once again, as Microsoft conceded earlier in the memo, the Internet is our poster child. Our best stop-thrust against embrace-and-extend is to point out that Microsoft is trying to close up the Internet. }

Netscape

In an attempt to renew it's credibility in the browser space, Netscape has recently released and is attempting to create an OSS community around it's Mozilla source code.

Organization & LIcensing

Netscape's organization and licensing model is loosely based on the Linux community & GPL with a few differences. First, Mozilla and Netscape Communicator are 2 codebases with Netscape's engineers providing synchronization.

- Mozilla = the OSS, freely distributable browser
- Netscape Communicator = Branded, slightly modified (e.g. hom epage default is set to home.netscape.com) version of Mozilla.

Unlike the full GPL, Netscape reserves the final right to reject / force modifications into the Mozilla codebase and Netscape's engineers are the appointed "Area Directors" of large components (for now).

Strengths

Capitalize on Anti-MSFT Sentiment in the OSS Community

Relative to other OSS projects, Mozilla is considered to be one of the most direct, near-term attacks on the Microsoft establishment. This factor alone is probably a key galvanizing factor in motivating developers towards the Mozilla codebase.

New credibility

The availability of Mozilla source code has renewed Netscape's credibility in the browser space to a small degree. As BharatS points out in http://ie/specs/Mozilla/default.htm:

{ The link to the BharatS quote is broken. }

"They have guaranteed by releasing their code that they will never disappear from the horizon entirely in the manner that Wordstar has disappeared. Mozilla browsers will survive well into the next 10 years even if the user base does shrink. "

Scratch a big itch

The browser is widely used / disseminated. Consequently, the pool of people who may be willing to solve "an immediate problem at hand" and/or fix a bug may be quite high.

Weaknesses

Post parity development

Mozilla is already at close to parity with IE4/5. Consequently, there no strong example to chase to help implicitly coordinate the development team.

Netscape has assigned some of their top developers towards the full time task of managing the Mozilla codebase and it will be interesting to see how this helps (if at all) the ability of Mozilla to push on new ground.

Small Noosphere

An interesting weakness is the size of the remaining "Noosphere" for the OSS browser.

1. The stand-alone browser is basically finished.

There are no longer any large, high-profile segments of the stand-alone browser which must be developed. In otherwords, Netscape has already solved the interesting 80% of the problem. There is little / no ego gratification in debugging / fixing the remaining 20% of Netscape's code.

2. Netscape's commercial interests shrink the effect of Noosphere contributions.

Linus Torvalds' management of the Linux codebase is arguably directed towards the goal of creating the best Linux. Netscape, by contrast, expressly reserves the right to make code management decisions on the basis of Netscape's *commercial / business*

interests. Instead of creating an important product, the developer's code is being subjugated to Netscape's stock price.

Integration Cost

Potentially the single biggest detriment to the Mozilla effort is the level of integration that customers expect from features in a browser. As stated earlier, integration development / testing is NOT a parallelizable activity and therefore is hurt by the OSS process.

In particular, much of the new work for IE5+ is not just integrating components within the browser but continuing integration within the OS. This will be exceptionally painful to compete aga inst.

Predictions

The contention therefore, is that unlike the Apache and Linux projects which, for now, are quite successful, Netscape's Mozilla effort will:

- Produce the dominant browser on Linux and some UNIX's
- Continue to slip behind IE in the long run

Keeping in mind that the source code was only released a short time ago (April '98), there is already evidence of waning interest in Mozilla. EXTREMELY unscientific evidence is found in the d ecline in mailing list volume on Mozilla mailing lists from April to June.

Mozilla Mailing List	April 1998	June 1998	% decline
Feature Wishlist	1073	450	58%
UI Development	285	76	73%
General Discussion	1862	687	63%

Internal mirrors of the Mozilla mailing lists can be found on http://egg.Microsoft.com/wilma/lists

{ Heh. The `egg' machine, it turns out, is a Linux box. }

Apache

History

Paraphrased from http://www.apache.org/ABOUT_APACHE.html

In February of 1995, the most popular server software on the Web was the public domain HTTP daemon developed by NCSA, University of Illinois, Urbana-Champaign. However, development of that httpd had stalled after mid-1994, and many webmasters had developed their own extensions and bug fixes that were in need of a common distribution. A small group of these webmasters, contacted via private e-mail, gathered together for the purpose of coordinating their changes (in the form of "patches"). By the end of February `95, eight core contributors formed the foundation of the original Apache Group. In April 1995, Apache 0.6.2 was released.

During May-June 1995, a new server architecture (code-named Shambhala) was developed which included a modular structure and API for better extensibility, pool-based memory allocation, and an adaptive pre-forking process model. The group switched to this new server base in July and added the features from 0.7.x, resulting in Apache 0.8.8 (and its brethren) in August.

Less than a year after the group was formed, the Apache server passed NCSA's httpd as the #1 server on the Internet.

Organization

The Apache development team consists of about 19 core members plus hundreds of web site administrators around the world who've submitted a bug report / patch of one form or another. Apache's bug data can be found at: http://bugs.apache.org/index.

A description of the code management and dispute resolution procedures followed by the Apache team are found on http://www.apache.org:

Leadership:

There is a core group of contributors (informally called the "core") which was formed from the project founders and is augmented from time to time when core members nominate outstanding contributors and the rest of the core members agree.

Dispute resolution:

Changes to the code are proposed on the mailing list and usually voted on by active members -- three +1 (yes votes) and no -1 (no votes, or vetoes) are needed to commit a code change during a release cycle

Strengths

Market Share!

Apache far and away has #1 web site share on the Internet today. Possession of the lion's share of the market provides extremely powerful control over the market's evolution.

In particular, Apache's market share in web server space presents the following competitive hurdles:

- Lowest common denominator HTTP protocol -- slows our ability to extend the protocol to support new applications
- Breathe more life into UNIX -- Where Apache goes, Unix must follow.

3rd Party Support

The number of tools / modules / plug-ins available for Apache has been growing at an increasing rate.

Weaknesses

Performance

In the short run, IIS soundly beats Apache on SPECweb. Moving further, as IIS moves into kernel and takes advantage deeper integration with the NT, this lead is expected to increase further.

Apache, by contrast, is saddled with the requirement to create portable code for all of its OS environments.

HTTP Protocol Complexity & Application services

Part of the reason that Apache was able to get a foothold and take off was because the HTTP protocol is so simple. As more and more features become layered on top of the humble web server (e.g. multi-server transaction support, POD, etc.) it will be interesting to see how the Apache team will be able to keep up.

ASP support, for example is a key driver for IIS in corporate intranets.

IBM & Apache

Recently, IBM announced it's support for the Apache codebase in its WebSphere application server. The actual result of the press furor is still unclear however:

- IBM still ships and supports both Apache and Domino's GO web server
- IBM's commitment appears to be:
  - Helping Apache port to strategic IBM platforms (AS/400, etc.)
  - Redistributing Apache binaries to customers who request Apache support
  - Support for Apache binaries (only if they were purchased through IBM?)
- IBM has developers actively participating in Apache development / discussion groups.
- IBM is taking a lead role in optimizing Apache for NT

Other OSS Projects

Some other OSS projects:

- **Gimp** -- http://www.gimp.org -- Gimp (GNU Image Manipulation Program) is an OSS project to create an Adobe Photoshop clone for Unix workstations. Feature-wise, however, their version 1.0 project is more akin to PaintBrush.
- **WINE / WABI** – http://www.wine.org -- Wine (Wine Is Not an Emulator) is an OSS windows emulation library for UNIX. Wine competes (somewhat) with Sun's WABI project which is non-OSS. Older versions of Office, for example, are able to run in WINE although performance remains to be evaluated.

{ This URL is wrong. See www.winehq.com. }

- **PERL** -- http://www.perl.org -- PERL (Practical Evaluation and Reporting Language) is the defacto standard scripting language for all Apache web servers. PERL is very popular on UNIX in particular due to its powerful text/string manipulation and UNIX's reliance on command line administration of all functionality.
- **BIND** --http://www.bind.org -- BIND (Berkeley Internet Name Daemon) is the de facto DNS server for the Internet. In many respects, DNS was developed on top of BIND.
- **Sendmail** -- http://www.sendmail.org – Sendmail is the #1 share mail transfer agent on the Internet today.
- **Squid** -- >http://www.squid.org – Squid is an OSS Proxy server based on the ICP protocol. Squid is somewhat popular with large international ISPs although it's performance is lacking.

{ This URL is wrong. See http://squid.nlanr.net. }

- **SAMBA** -- http://www.samba.org -- SAMBA provides an SMB file server for UNIX. Recently, the SAMBA team has managed to reverse engineer and develop an NT domain controller for UNIX as well. SGI employs one of the SAMBA leads. http://www.sonic.net/~roelofs/reports/linux-19980714-phq.html: "*By the end of the year ... Samba will be able to completely replace all primary NT Server functions.*" { The Samba URL is wrong. See http://samba.org.au. }
- **KDE** -- http://www.kde.org -- "K" Desktop Environment. Combines integrated browser, shell, and office suite for Unix desktops. Check out the screen shots at: http://www.kde.org/kscreenshots.html and http://www.kde.org/koffice/index.html.
- **Majordomo** -- the dominant mail list server on the Internet is writtenentirely in PERL via OSS.

Microsoft Response

In general, a lot more thought/discussion needs to put into Microsoft's response to the OSS phenomena. The goal of this document is education and analysis of the OSS process, consequently in this section, I present only a very superficial list of options and concerns.

Product Vulnerabilities

Where is Microsoft most likely to feel the "pinch" of OSS projects in the near future?

Server vs. Client

The server is more vulnerable to OSS products than the client. Reasons for this include:

- **Clients "task switch" more often** -- the average client desktop is used for a wider variety of apps than the server. Consequently, integration, ease-of-use, fit & finish, etc. are key attributes.
- **Servers are more task specific --** OSS products work best if goals/precedents are clearly defined -- e.g. serving up commodity protocols
- **Commodity servers are a lower "commitment" than clients** -- Replacing commodity servers such as file, print, mail-relay, etc. with open source alternatives

doesn't interfere with the end-user's experience. Also, in these commodity services, a "throw-away" "experimental" solution will often by entertained by an organization.

o **Servers are professionally managed --** This plays into OSS's strengths in customization and mitigates weaknesses in lack of end-user ease of use focus.

Capturing OSS benefits -- Developer Mindshare

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 our own evangelization efforts appear to scale.

{ That is, Microsoft is being both out-thought and out-marketed by Open Source -- **and knows it**! }

How can Microsoft capture some of the rabid developer mindshare being focused on OSS products?

Some initial ideas include:

o **Capture parallel debugging benefits via broader code licensing --** Be more liberal in handing out source code licenses to NT to organizations such as universities and certain partners.

o **Provide entry level tools for low cost / free** -- The second order effect of tools is to generate a common skillset / vocabulary tacitly leveraged by developers. As NatBro points out, the wide availability of a consistent developer toolset in Linux/UNIX is a critical means of implicitly coordinating the system.

o **Put out parts of the source code** -- try to generate hacker interest in adding value to MS-sponsored code bases. Parts of the TCP/IP stack could be a first candidate. OshM points out, however that the challenge is to find some part of MS's codebase with a big enough Noosphere to generate interest.

o **Provide more extensibility** -- The Linux "enthusiast developer" loves writing to / understanding undocumented API's and internals. Documenting / publishing some internal API's as "unsupported" may be a means of generating external innovations that leverage our systems investments. In particular, ensuring that more components from more teams are scriptable / automatable will help ensure that power users can play with our components.

{ How curious. This paragraph only makes sense if Microsoft has "undocumented internal APIs" to document. Didn't Microsoft executives testifying in a federal restraint-of-trade lawsuit deny this under oath in 1995? I wonder if perjury charges might be in order... A former Microserf tells me that Microsoft departments see themselves almost as separate organizations. Parallel (and competitive) software development spurs both groups onward. The 'surviving' product is then what MS releases. This internal adversarial approach is taken so far that many crucial components do not have documented APIs -- primarily to ensure that the Dev team is not broken up and moved to other projects. MS is protected against perjury charges by the simple fact that their APIs are not even documented for internal MS use, so they are not holding anything back from competitors. }

o **Creating Community/Noosphere**. MSDN reaches an extremely large population. How can we create social structures that provide network benefits leveraging this huge developer base? For example, what if we had a central VB showcase on Microsoft.com which allowed VB developers to post & published full source of their VB projects to share with other VB developers? I'll contend that many VB developers would get extreme ego gratification out of having their name / code downloadable from Microsoft.com.

o **Monitor OSS news groups**. Learn new ideas and hire the best/brightest individuals.

Capturing OSS benefits -- Microsoft Internal Processes

What can Microsoft learn from the OSS example? More specifically: How can we recreate the OSS development environment internally? Different reviewers of this paper have consistently pointed out that internally, we should view Microsoft as an idealized OSS community but, for various reasons do not:

o **Different development "modes".** Setting up an NT build/development environment is extremely complex & wildly different from the environment used by the Office team.

o **Different tools / source code managers.** Some teams use SLM, other use VSS. Different bug databases. Different build processes.

o **No central repository/code access.** There is no central set of servers to find, install, review the code from projects outside your immediate scope. Even simply providing a central repository for debug symbols would be a huge improvement. NatBro:

"a developer at Microsoft working on the OS can't scratch an itch they've got with Excel, neither can the Excel developer scratch their itch with the OS -- it would take them months to figure out how to build & debug & install, and they probably couldn't get proper source access anyway"

o **Wide developer communication**. Mailing lists dealing with particular components & bug reports are usually closed to team members.

o **More component robustness**. Linux and other OSS projects make it easy for developers to experiment with small components in the system without introducing regressions in other components: DavidDs:

"People have to work on their parts independent of the rest so internal abstractions between components are well documented and well exposed/exported as well as being more robust because they have no idea how they are going to be called. The linux development system has evolved into allowing more devs to party on it without causing huge numbers of integration issues because robustness is present at every level. This is great, long term, for overall stability and it shows."

The trick of course, is to capture these benefits without incurring the costs of the OSS process. These costs are typically the reasons such barriers were erected in the first place:

o **Integration.** A full-time developer on a component has a lot of work to do already before trying to analyze & integrate fixes from other developers within the company.

o **Iterative costs & dependencies.** The potential for mini-code forks between "scratched' versions of the OS being used by one Excel developer and "core" OS used by a different Excel developer.

Extending OSS benefits -- Service Infrastructure

Supporting a platform & development community requires a lot of *service* infrastructure which OSS can't provide. This includes PDC's, MSDN, ADCU, ISVs, IHVs, etc.

The OSS communities "MSDN" equivalent, of course, is a loose confederation of web sites with API docs of varying quality. MS has an opportunty to really exploit the web for developer evangelization.

Blunting OSS attacks

Generally, Microsoft wins by attacking the core weaknesses of OSS projects.

De-commoditize protocols & applications

OSS projects have been able to gain a foothold in many server applications because of the wide utility of highly commoditized, simple protocols. By extending these protocols and developing new protocols, we can deny OSS projects entry into the market.

David Stutz makes a very good point: in competing with Microsoft's level of desktop integration, "*commodity protocols actually become the means of integration*" for OSS projects. There is a large amount of IQ being expended in various IETF working groups which are quickly creating the architectural model for integration for these OSS projects.

{ In other words, open protocols must be locked up and the IETF crushed in order to ``de-commoditize protocols & applications'' and stop open-source software.

A former Microserf adds: only half of the reason MS sends people to the W3C working groups relates to a desire to improve RFC standards. The other half is to give MS a sneak peak at upcoming standards so they can "extend" them in advance and claim that the `official' standard is `obsolete' when it emerges around the same time as their `extension'.

Once again, open-source advocates' best response is to point out to customers that when things are ``de-commoditized'', vendors gain and customers lose. }

Some examples of Microsoft initiatives which are extending commodity protocols include:

o **DNS integration with Directory**. Leveraging the Directory Service to add value to DNS via dynamic updates, security, authentication

o **HTTP-DAV**. DAV is complex and the protocol spec provides an infinite level of implementation complexity for various applications (e.g. the design for Exchange over DAV is good but certainly not the single obvious design). Apache will be hard pressed to pick and choose the correct first areas of DAV to implement.

{ What wonderful, scathing irony! Four days after Halloween I hit the net, Greg Stein (an ex-Microserf, no less) announced working HTTP-DAV support for Apache as open-source software. }

o **Structured storage**. Changes the rules of the game in the file serving space (a key Linux/Apache application). Creates a compelling client-side advantage which can be extended to the server as well.

o **MSMQ for Distributed Applications**. MSMQ is a great example of a distributed technology where most of the value is in the services and implementation and NOT in the wire protocol. The same is true for MTS, DTC, and COM+.

Make Integration Compelling – Especially on the server

The rise of specialty servers is a particularly potent and dire long term threat that directly affects our revenue streams. One of the keys to combating this threat is to create integrative scenarios that are valuable on the server platform. David Stutz points out:

The bottom line here is whoever has the best network-oriented integration technologies and processes will win the *commodity server business*. There is a convergence of embedded systems, mobile connectivity, and pervasive networking protocols that will make the number of servers (especially "specialist servers"??) explode. The general-purpose commodity client is a good business to be in - will it be dwarfed by the special-purpose commodity server business?

o **System Management**. Systems management functionality potentially touches all aspects of a product / platform. Consequently, it is not something which is easily grafted onto an existing codebase in a componentized manner. It must be designed from the start or be the result of a conscious re-evaluation of all components in a given project.

o **Ease of Use**. Like management, this often must be designed from the ground up and consequently incurs large development management cost. OSS projects will consistently have problems matching this feature area

o **Solve Scenarios**. ZAW, dial up networking, wizards, etc.

o **Client Integration**. How can we leverage the client base to provide similar integration requirements on our servers? For example, MSMQ, as a piece of middleware, requires closely synchronized client and server codebases.

o **Middleware control is critical**. Obviously, as servers and their protocols risk commoditization higher order functionality is necessary to preserve margins in the server OS business.

Organizational Credibility

o **Release / Service pack process.** By consolidating and managing the arduous task of keeping up with the latest fixes, Microsoft provides a key customer advantage over basic OSS processes.

o **Long-Term Commitments.** Via tools such as enterprise agreements, long term research, executive keynotes, etc., Microsoft is able to commit to a long term vision and create a greater sense of long term order than an OSS process.

Other Interesting Links

- http://www.lwn.net/ -- summarizes the weeks events in Linux development world.
- Slashdot -- http://slashdot.org/ -- daily news / discussion in the OSS community
- http://www.linux.org
- http://www.opensource.org
- http://news.freshmeat.net/ -- info on the latest open source releases & project updates

Acknowledgments

Many people provided, datapoints, proofreading, thoughtful email, and analysis on both this paper and the Linux analysis:

Nat Brown

Jim Allchin

Charlie Kindel

Ben Slivka

Josh Cohen

George Spix

David Stutz

Stephanie Ferguson

Jackie Erickson

Michael Nelson

Dwight Krossa

David D'Souza

David Treadwell

David Gunter

Oshoma Momoh

Alex Hopman

Jeffrey Robertson

Sankar Koundinya

Alex Sutton

Bernard Aboba

Halloween II

COLOR CODING NOTE:
{green} = comments by Eric Raymond
red = original text, highlighted by Eric Raymond
black = original text

Vinod Valloppillil (VinodV)
Josh Cohen (JoshCo)
Aug 11, 1998 - v1.00

Microsoft Confidential

Table of Contents *

Linux Operating System

The Next Java VM?

Executive Summary

The Linux OS is the highest visibility product of the Open Source Software (OSS) process. Linux represents a best-of-breed UNIX, that is trusted in mission critical applications, and - due to it's open source code - has a long term credibility which exceeds many other competitive OS's.

Linux poses a significant near-term revenue threat to Windows NT Server in the commodity file, print and network services businesses. Linux's emphasis on serving the hacker and UNIX community alleviates the near-medium term potential for damage to the Windows client desktop.

In the worst case, Linux provides a mechanism for server OEMs to provide integrated, task-specific products and completely bypassing Microsoft revenues in this space.

[*This document assumes that the reader has read the "Open Source Software" doc first. Many of the ideas / assertions here are derived from the previous doc and many other applicable Open Source arguments are not repeated here for brevity.*]

Linux History

What is it?

Linux (pronounced "LYNN-ucks") is the #1 market share Open Source OS on the Internet. Linux derives strongly from the 25+ years of lessons from the UNIX operating system.

Top-Level Features:

- Multi-user / Multi-threaded (kernel & user)
- Multi-platform (x86, Alpha, MIPS, PowerPC, SPARC, etc.), source compatibility
- Protected 32-bit memory space for apps; Virtual Memory support;
- 64-bit support (platform dependent)
- SMP (Intel & Sun CPU's)
- Supports multiple file systems (FAT16, FAT32, NTFS, various UNIX)
- High performance networking
  - NFS/SMB/IPX/Appletalk networking
  - Fastest stack in Unix vs. Unix performance tests
- Disk Management
  - Striping, mirroring, RAID 0,1,5
- Xfree86 GUI

History

An excellent piece on the history of the Linux Operation system is provided by Wired Magazine at http://www.wired.com/wired/5.08/linux.html. I've paraphrased some of the key points below.

Linux was original the brainchild of Linus Torvalds, an undergraduate student at the University of Helsinki. In addition to a 80386-based kernel, Linus wrote keyboard and screen drivers to attach to PC hardware and provided this code under GNU's Public License on an FTP site in the summer of 1991.

After hosting his work on the FTP site, he announced it's availability on a Minix USENET discussion group in late summer 1991. By January of 1992, over 100 users / hackers had downloaded Linux and - more importantly - were regularly contributing / updating the source code with new fixes, device drivers, etc.

In contrast to the FSF/GNU work, which provided developers an open source abstraction above the underlying, commercial UNIX OS kernel, Linux's team was creating a completely open source kernel. In time, more and more of the GNU user/shell work was ported to Linux to round out the platform for hackers.

Forbes magazine's story on Linux has some excellent data on Linux's development history http://www.forbes.com/forbes/98/0810/6209094s1.htm:

{ This URL is stale. See http://www.forbes.com/forbes/98/0810/6203094s1.htm .

Date	Users	Version	Size (LOC)
1991	1	0.01	10k
1992	1000	0.96	40k
1993	20000	0.99	100k
1994	100000	1.0	170k
1995	500000	1.2	250k
1996	1.5M	2.0	400k
1997	3.5M	2.1	800k
1998	7.5M	2.1.110	1.5M

The LOC count appears to be inclusive of all Linux ports including x86, PPC, SPARC, etc.

Linux 1.0 - March 1994

Linux 1.0 was the first major release and led to the creation of "distributions." Prior to 1.0, linux existed as a piecemeal kernel with no centralized place to get a full working OS.

Major Features:

- Virtual Memory Management / memory Mapping / Buffer cache
- Job Control
- Device support for popular Network Cards, Hard Drives, CDROMs, etc
- Named Pipes, IPC
- Original EXTFS support instead of Minixfs
- Preemptive multitasking

Management Structure

After the release of version 1.0. The Linux developer community adopted a management structure to control what is added to the kernel with even numbered releases as stable, production release branches and odd numbered versions were "developer" branches.

While major areas of the kernel have "owners" which maintain their areas, Linus remains the final say on what does and does not go into the kernel. In large part, this structure remains in place.

It is important to distinguish that this management structure only controls the actual kernel and does not include supporting areas like the GUI, system utilities and servers, and system libraries.

Since 1.0, the following 1.x branches existed:

1.1 3/95

1.2 8/95

1.3 6/96

Version 1.3 evolved to become version 2.0

Linux 2.0 - June 1996

Linux v2.0 was the first major release could effectively compete as a UNIX distribution. The kernel, system libraries, the GNU Unix tool, X11, various open source server applications such as BIND and sendmail, etc. were frozen and declared part of Linux 2.0.

Around the same time the GNU/FSF agreed, reluctantly, to make the Linux kernel the official kernel of the GNU operating system.

{ No, FSF did not so agree. It's still working on its own ``HURD'' kernel. }

Some of the new base libraries and tools:

- Kernel modules 2.0.0 - Basic kernel module support
- PPP daemon 2.2.0f - Dialup networking
- Dynamic linker (ld.so) 1.7.14 - Shared libraries
- GNU CC 2.7.2 - C compiler, tools, and debugger
- Binutils 2.6.0.14 - Support for various binary executable formats
- Linux C Library Stable: 5.2.18,
- Linux C++ Library 2.7.1.4
- Termcap 2.0.8 - Console mode terminal drivers
- Procps 1.01 - ProcFS file system maps kernel objects to the filesystem
- SysVinit 2.64 - A system V boot system, SYSV compliant named pipes.
- Net-tools 1.32-alpha- Basic Networking tools such as telnet, finger, etc
- Kbd 0.91 - Cons ole mode keyboard/scrollback/ virtual screens support

Subsequent Versions

The current 2.0.x stable version is 2.0.34, which was released in May 1998. Prior to this, 2.0.33 was released in Dec 1997. The current development branch is 2.1.108 (as of July 14, 1998).

Process Slowdown

With the growth of the kernel, Linux's release frequency has slowed measurably. There is growing frustration about when 2.2, the next "stable release" version will ship. The sheer size of the codebase has begun to overrun the resources of Linus. There is a backlog of patches to be merged and often, Linus is becoming the choke point.

The current release tree, 2.0.x has iterated 34 versions in 2 years. The development branch, 2.1.x, which will eventually become 2.2 has been going on since 9/96 spanning 108 versions and no ship date in sight.

{ That's true. On the other hand, most people have been using 2.1.x for many months with no crashes, and fewer reboots over that time than the average winows or NT installation has to do in an average week.

Linus could have shipped 2.2 in Spring '98 and had a stable, high-quality kernel. It's just that he's got higher standards than Microsoft. }

Even though the feature freeze is declared, major changes continue to get integrated into the kernel. Most merges seem to be due to fundamental bug fixes and or cross platform issues.

Organization

An analysis / description of the OSS development organization and process is in a second memo titled "Open Source Software." This section describes attributes of OSS that are unique to Linux.

Wired Magazine ran a recent story chronicling the history of Linux "The Greatest OS that (N)ever was" http://www.wired.com/wired/5.08/linux.html.

The growth of the development team mirrored the organic, not to say chaotic, development of Linux itself. Linus began choosing and relying on what early Linux hacker Michael K. Johnson calls "a few trusted lieutenants, from whom he will take larger patches and trust those patches. The lieuts more or less own relatively large pieces of the kernel."

As with other OSS projects, the General Public License ("CopyLeft") and it's relatives are considered instrumental towards creating the dynamic behavior around the Linux codebase:

In a sense, GPL provided a written constitution for the new online tribe of Linux hackers. The license said it was OK to build on, or incorporate wholesale, other people's code - just as Linux did - and even to make money doing so (hackers have to eat, after all). But you couldn't transgress the hacker's fundamental law of software: source code must be freely available for further hacking

Linux Technical Analysis & OS Structure

Anatomy of a Distribution

"Linux" is technically just a kernel, not the entire supporting OS. In order to create a usable product, Linux "distributions" are created which bundle the kernel, drivers, apps and many other components necessary for the full UNIX/GUI experience.

These subsystems are typically developed in an OSS manner as well and several of them - e.g. the Xfree86 GUI - have a codebase size/complexity that exceeds the Linux kernel.

These external components come from many sources and are individually hand picked by the distribution vendor for a particular product. A frequent source of controversy stems from distribution vendors bundling non-GPL code with the Linux kernel and mass distributing them.

A partial list of components is in the following table:

Component	Codebase / Name	Provider/Maintainer(s)
Kernel	Basic OS, Networking Stack	Linux (http://www.kernel.org)
File System(s)	Msdos, ext2fs	Linux Kernel
Sys Libs	Glibc, Lib5c	GNU / FSF
Drivers		Linux, Individual Contributors
User Tools	Gnu user tools	GNU/FSF
System Installation	LISA	Caldera
App Install Management	RedHat Package Manager	RedHat
Development Tools	GNU Development tools GCC	GNU/FSF
Web Server	APACHE	The Apache Group http://www.apache.org/
Mail Server	SendMail	http://www.sendmail.org
DNS Server	BIND	http://www.bind.org
SMB Server	SAMBA	http://www.samba.org
X Server	Xffree86 / MetroX	Xfree86 project / MetroX commercial
Window Manager	FVWM	GPL
Widgets	Motif	X Consortium
Desktop Tools	X Contrib KDE Gnome	X Consortium http://www.kde.org http://www.gnome.org
Management	RPM Package Installed Roll own distribution specific	RedHat (free) Debian / Slackware

Descriptions of some of the larger components are below:

Kernel - GPL

The kernel is the core part of Linux that is expressly managed by Linus and his lieutenants and is protected via the GPL.

Functions contained in the Linux Kernel include:

- Core OS Features (scheduling, memory management, threads, Hardware Abstraction, etc)
- Network Stack
- File system

Extensive on-line documentation of the Linux kernel architecture and components can be found on: http://sunsite.unc.edu/linux/LDP/tlk/tlk.html. Note that video drivers exist outside the kernel - the kernel only has rudimentary text display support to a console.

Drivers -- GPL

An assortment of modules for standard functions and devices are typically part of the kernel distribution. In addition, a selection of non-standard modules is often included.

Mostly GPL, however in some cases, NDAs with hardware manufacturers are required to get specs to make a driver, in which case they are not open source.

Linux device drivers are typically developed by users for specific devices on their machines. This incremental, piecemeal process has created a very large pool of device drivers for Linux (as of 7/1/98):

- Video: http://sunsite.unc.edu/LDP/HOWTO/Hardware-HOWTO-6.html -- close to **400** drivers available
- Network: http://sunsite.unc.edu/LDP/HOWTO/Hardware-HOWTO-11.html -- **~75** network cards supported
- PCMCIA http://sunsite.unc.edu/LDP/HOWTO/Hardware-HOWTO-26.html – **~150** supported cards.

NatBro points out:

An important attribute to note which has led to volume drivers is the ease with which you can write drivers for linux, and the relatively powerful debugging infrastructure that linux has. Finding and installing the DDK, and trying to hook up the kernel debugger and do any sort of interaction with user-mode without tearing the NT system to bits is much more challenging than writing the simple device-drivers for linux. Any idiot could write a driver in 2 days with a book like "Linux Device Drivers" -- there is no such thing as a 2-day device-driver for NT

{ A Microsoft developer who wishes to remain anonymous comments:

MS really shoots themselves in the foot in driver space. For those of us that will willingly spend effort to improve the quality, functionality, and availability of device drivers for MS OSs, the tools, etc. they provide are actually getting progressively worse. This is happening largely because MS fails to grasp the basic concepts you've been talking about. In response to driver quality and availability issues, they've consistently made exactly the wrong decisions: rather than encouraging more people to help and providing better tools, they try to make the system more and more closed and dumb down the tools.

A significant side effect of this is the killing of innovation - MS, being a monolithic organization, can really only concentrate on a few initiatives at a time, and improving device (printers, video, input, disk, ...) functinality and performance is rarely very high on the list. This is a great example of where they should let someone else (either a partner or an industry standards body - or maybe even an open-source community!) `homestead' a driver space. Instead they intentionally make it more difficult for these 3rd parties to contribute. The reasonably plausible reason MS gives for this is to improve system stability - they blame a lot of NT robustness issues on ``bad 3rd party drivers''. Instead of helping the 3rd parties build better drivers (better tools, better tech support/documentation, actually soliciting input from the 3rd parties, etc.), though, the MS approach is to assert more control of the driver space by making more of the system closed.

The ironic effect of this, though, is that the 3rd parties still have the business need to innovate despite the MS restrictions, so they end up frequently hacking around the closed parts of the MS solutions with no documentation - and the overall system stability goes down, not up. By making the system more closed, they are decreasing stability & quality, slowing innovation, and losing the opportunity to take advantage of 3rd parties willing to contribute for free. The desire by MS to own and control the interfaces is so strong, though, that they can't step back far enough to understand what they are doing.

}

Recently, a small number of hardware vendors have begun to provide Linux drivers for their NICs (3Com) and SCSI adapters (Adaptec). These drivers are believed to be protected by the Library-GPL and are consequently not open source (the Library-GPL is described later). It remains to be seen whether this will create the momentum to develop more commercial drivers for Linux.

System Libraries & Apps - GNU GPL

System libraries provide:

- Basic POSIX api's for system services
- Basic API's to support commandline / shell utilities.

The system libraries in a Linux distribution are NOT managed by Linus. As such, there has been a small amount of versioning / forking in this area with two dominant libraries - glibc and lib5c which introduce minor incompatibilities between different apps.

User Tools (GPL, GNU FSF)

These are basic UNIX command line tools and shell environments. Many shell environments exist although all are supported by the FSF.

Also included in this category are "old standby" apps such as finger, telnet, etc.

Development Tools (GPL)

A hallmark of the UNIX operating system is the free availability of development tools / compilers. The GCC and PERL language compilers are often provided for free with all versions of Linux and are available for other UNIXes as well.

These tools are the "old standbys" of the UNIX development world and are widely used across all Unix platforms. This mass commoditization of development/debug tools is a key contributor to the common skillset efficiencies realized by the Linux process.

By the standards of the novice / intermediate developer accustomed to VB/VS/VC/VJ, these tools are incredibly primitive.

GUI / UI

X Server

The X Server standard is owned by MIT under contract by the X Consortium. X Consortium's licensing practices are viewed as too restrictive by the OSS crowd so a series of public X initiatives were launched with Xfree86 being the dominant distribution.

Interestingly, the Xfree86 development team licenses their code under the BSD license because they consider GPL too restrictive: http://www.redhat.com/linux-info/xfree86/developer.html.

Configuring the XFree86 system on Linux can be a very difficult, time consuming process. Linux has no hardware abstraction layer for video services, and most video card manufacturers do not provide Linux OS video drivers. Thus, XFree86 provides internal support for a wide variety of video cards and chipsets. Correctly configuring XFree86 requires the user to know the manufacturer, model, and chipset for their video card. In many cases, the user must know or calculate the video timings as well.

Widgets & Desktops

There are multiple widget sets which exist in many applications, so all X applications do not look the same or act the same ways like in Windows. Motif is considered the defacto Unix widget set, but since it is not freely distributable, it is contrary to the Linux model.

Consequently, Linux distributions usually choose one of several similar, but not completely compatible Widget sets.

- Motif
- LessTif
- Xaw3d (3d athena widgets that look like motif)
- QT

Obviously, this mess has spawned several efforts to unify the "desktop" as well as the widget sets. In typical Linux fashion, there are several competing efforts:

- Gnome/totally new
- KDE
- FreeQT/KDE
- CDE/commercial

Commercial Linux OS

Binary Compatibility

Server

Almost all of the system components necessary to run server applications are part of the core distribution maintained by Linus. Consequently, for a given hardware type, almost all Linux server application binaries will natively run. Across hardware types (e.g. x86 vs. PPC), generally only a recompile of the application is necessary.

There is essentially 100% source code compatibility for system application code.

Solaris / SCO x86 Compatibility

Via compatibility libraries, Linux on x86 is able to natively execute most SCO UNIX and Solaris x86 binaries. Oracle on SCO is widely cited as an example (although Oracle does not "officially" support SCO binaries on top of Linux - also Oracle has recently announced development of a native Linux version of Oracle 8 to ship in March 1999)

Client

Client distributions, however, are a different story stemming most directly from the current "mess" in X-windows / GUI systems for Linux.

Binary compatibility issues generally stem from differences in non-kernel code that's required to turn the kernel into a full OS.

Binary Incompatibility: Netscape Communicator

One example of this incompatibility is Netscape Communicator for Linux. The released versions of Netscape Communicator for Linux are built based on libc5, instead of the newer glibc which Caldera supports. RedHat, however ships glibc instead of libc5 requiring users install libc5 as well as glibc.

RedHat

http://www.redhat.com

RedHat Corporation was founded in 1995 by a pair of Linux developers/enthusiasts with the intent of creating a commercially supported, "cleaned-up" Linux distribution.

The company currently has ~35 employees. Financials and some run-rate information is available in an interview with their CEO in Infoworld (http://www.infoworld.com/cgi-bin/displayArchive.pl?/98/23/e03-23.102.htm_:

Bob Young, president of Red Hat expects the 3-year old company to earn revenues of $10 million this year and to ship about 400,000 copies of Linux, ranging from $50 to near $1,000 for a supported version.

Commercially-Developed Extensions

Perhaps the most interesting aspect of Red Hat's business model is their extremely active and continuing contributions to the Linux community. Several prior initiatives spearheaded by RedHathave been released as OSS for modification. In most cases, these code releases were simple fixes or additional drivers.

Redhat actively employs several key Linuxdevelopers and pays them to hack Linux fulltime. Some of the components which have been "donated" back to the Linux effort include:

- **RedHat Package Manager** - RPM is Linux component which provides application install / maintenance facilities for Linux similar to the Application Manifest being developed by Microsoft.

- **Pluggable-Authentication Manager** - PAM is similar to the NT SSPI / SAM system and allows for componentized plug-ins to handle the authentication function (RedHat provides an LDAP plugin). PAM was originally available on Sun systems.

One ofthe larger "grants" however has been the now universal "Redhat Package Manager" or RPM which ships with almost all Linux distributions. RPM creates the concept of an application manifest which simplifies the job of installing & removing applications on top of Linux.

Redhat's current development project is a new GUI for Linuxcall "Gnome". Gnome is a response to latent concerns with non-GPL versions of the X-windows user interface.

Product Features

Of the Commercial LinuxDistributors, Redhat has the largest array of SKU's. At the highest end, Redhat bundles the following with their distributions of Linux:

- Apache Web Server
- Corel WordPerfect
- DBMaker DBMS by Casemaker
- Xfree86 window server

Caldera

Caldera is Ray Noorda's latest company with its eye on the operating system marketplace. Caldera's financials and sales are unpublished but it is widely believed to be the #2 commercial Linux vendor after RedHat.

Caldera bundles several components with their version of Linux including:

- StarOffice 4.0 by Germany's Star Corp.
- Adabas SQL Server by Software AG
- Netware client & Admin
- Netscape fasttrack server + communicator
- Xfree86 and MetroX X-window systems

Others

Other Linux distributions seem to be falling by the wayside of RedHat and Caldera. They include SlackWare, SuSe, and Debian to name a few. A comprehensive list of distributions can be found on http://www.linux.org.

{ Writing off SuSE displays ignorance. They're #1 in Linux-happy Europe, and could become a threat to NT Workstation in the near future. }

Commercial Linux ISV's

There are currently no major ISV's who derive a significant percentage of their sales from the Linux platform. A somewhat complete list of the commercial apps available on Linux can be found on: http://www.uk.linux.org/LxCommercial.html.

Reasons for this include:

- **First-use Linux apps are free** - most of the primary apps that people require when they move to Linux are already available for free. This includes web servers, POP clients, mail servers, text editors, etc.

- **Linux market is still immature** - the Linux market is still in its infancy and the current state of Linux commercial software may change radically in the coming months

- **Current Linux users are wary of commercial products** - you can scout any of various Linux discussion and mailing lists and quickly run into users admonishing commercial software providers and trying to launch a jihad against category X via open source software (at the time of this writing, Lotus Notes is a popular target)

Library-GPL

Unlike the GPL (General Public License - described in depth in "Open Source Software") which forces all derivative works to be free, Linuxsoftware libraries have the more limited "Library GPL" which allows applications which merely link to Linux to be considered non-derivative.

The Library-GPL removes a key impediment to commercial software vendors developing products on top of Linux.

The Library-GPL is defined at http://www.fsf.org/copyleft/lgpl.html

Binary Unix Compatibility

Linux adheres to several UNIX standards most notably POSIX 1003.1c. When compiled and running on it's various CPU platforms, Linux is generally binary compatible (more so on the server than on the desktop) with the primary commercial UNIXs including:

- Solaris/SunOS on SPARC
- Solaris on x86
- SCO on x86
- Digital UNIX on Alpha
- SGI IRIX on MIPs

Microsoft

Microsoft's current involvement in Linux is limited to distribution of client code for strategic services such as Netshow as well as helping SAG port DCOM to Linux. IE is currently not officially supported on Linux.

Intel

Intel is directly involved in helping port Linux to Merced. Intel is also involved with the GCC over Merced development efforts.

Netscape

In the press, Netscape is sited as the #1 commercial provider of software for Linux. Marc Andreeson has been extensively quoted as saying that "Linux is a tier 1 platform for Netscape".

Until recently, however, the only server product that Netscape explicitly sells for Linux is their Fasttrack server with other servers merely being licensed to the respective Linux vendors for their own redistribution. On July 21st, however, Netscape formally announced intentions to port all of their server application products to Linux starting with Mail and Directory services.

All of Netscape's client products are available on the Linux platform.

Oracle

Oracle recently announced (7/18/98) their support for Oracle 8 on top of Linux to be shipped in March 1999.

{ Oracle 8.0.5 for Linux has been shipped. In fact, Oracle is giving away copies free for development use. }

Sun

Sun's involvement in Linux is inconclusive. Early this year (1998), Sun joined the board of Linux International which is one of many user groups representing Linux.

At one level, Linux competes (quite favorably) against Sun's own Solaris x86 port.

At a secondary level, Sun may view Linux as a strategic ally b/c it generally represents the low-end of the software market and could therefore arguably hurt Microsoft more than it hurts Sun.

SoftwareAG

SoftwareAG has ported it's ADABAS database server to Linux and is currently bundled with Caldera's distribution.

Corel

Corel has ported their WordPerfect Suit to Linux and is currently offering it bundled with several of RedHat's SKU's

Computer Associates

Recently announced intentions to port CA-Ingres DB to Linux: http://x10.dejanews.com/getdoc.xp?AN=370037691&CONTEXT=900053229.949289093&hitnum=0.

Market Share

Linux's exact market share is very difficult to calculate because:

- The majority of Linux installations are downloaded from anonymous FTP sites - NOT purchased. Consequently, there are no published sales figures to track.
- (Some) Commercial Linux purchases can be used to install multiple machines
- Because Linux revs so often, there's a very high likelihood of double-counting actual installations vs. downloads/purchases
- There are no separate client & server distributions. Consequently, it's difficult to compare Linux numbers wholesale to NTS / NTW numbers without accurate usage data from the Linux community.

Below I include data / pointers from some of the more prominent attempts to isolate the number of Linux users.

Installed Base

The most comprehensive Linux market share survey was published by Red Hat in March 1998: http://www.redhat.com/redhat/linuxmarket.html

Using available data collected from other distributions, RedHat calculated a retail CD sell rate of :

{ The link above has changed to http://www.redhat.com/knowledgebase/linuxmarket.html.. }

- 1996: **450,000**
- 1997: **750,000**

RedHat's estimate of the growth of the Linux installed user base (which includes CD purchases as well as downloads as well as clent + server) is:

- 1993: **100k**
- 1994: **500k**
- 1995: **1.5M**
- 1996: **3.5M**
- 1997: **7.5M**

Other estimates put the Linux installed base from 5 Million (Ziff Davis), to 10 Million (Linux advocates).

Server

IDC's most recent "Server Operating Environments" report provides the following breakdown of shipments in the Server OS space.

Using the 240K number shipped in 1997, IDC seems to be estimating ~750K total installed Linux *server* systems. Compared to other market share studies, IDC's may be underestimating the actual new Linux server installations - I believe IDC may be counting only top distributions in their survey.

Client

Starting with Dataquest's market share figures published in June '98, I injected the incremental Linux numbers derived from RedHat's market survey (showing 7.5M users at the end of 1997).

Distributor Market Share

IDC provides information on the relative market share of the Linux distributors:

Linux Qualitative Assessment

I purchased and installed a copy of Caldera's OpenLinux v1.2 standard edition. I installed it on an old P5-100 / 32MB RAM machine in my office that used to run NT4. Knowing that device driver support on Linux was well below NT's, I intentionally chose a machine and peripherals that represented the 80% of the *installed base* (e.g. 3c509 NIC, Adaptec SCSI controller, etc.)

{ VinodV is confused. Stock Linux has a much broader range of supported drivers than NT. (claim documented at the Red Hat site). Interestingly, he later contradicts himself on this score. }

Installation

Caldera provided an auto-run CD which launched directly into their setup program - "LISA". Lisa prompted me for:

- Language selection (an interesting future research project would be to truly understand the depth of localization support provided by Linux.)
- Keyboard selection
- IDE hard drive & CD ROM device detection.

Although the dialogs could use a lot of work (e.g. many questions were phrased as double-negatives - "Should setup disable plug & play device detection (yes/no)"), up to this point I was asked no questions that a power user couldn't correctly answer.

A second round of device detection impressively auto-discovered my:

- Adaptec SCSI adapter
- Plextor CD-ROM drive
- Seagate Hard Drive
- 3Com 3c509 Ethernet adapter

I selected default device settings for each hardware option, selected "typical" install options and then LISA started copying.

This phase of the install/setup process was finished in 30 minutes (most of that time copying) and with a total of ~15 dialog boxes.

UI

As mentioned earlier, one of the quirks of UNIX / Linux relative to NT is that video drivers run in userspace and are not required for most system functionality. Linux is quite content with just a command prompt.

A second round of installation scripts was necessary to install the GUI. The installer gave me the option of choosing which video subsystem to install / configure and I chose the Xfree86 server because it's an entirely open source system (the other option - MetroX - was provided by Caldera and is believed to be the more stable codebase).

This part of setup definitely required knowledge of video systems even beyond many power users. Not only did I have to know the name / make / model of my video card and chipsets but I was presented with questions about their revision numbers, the scan rates of my monitors, etc. After significant trial and error, I finally got my video system working correctly.

The latest generation Xfree86 + CDE was slick and definitely represented among the best-of-breed in UNIX GUI's. A SUN desktop user would be perfectly at home here. *An advanced Win32 GUI user would have a short learning cycle to become productive.*

Following UNIX philosophy, however, mastery of the GUI was not enough to use the full system. Simple procedures such as reading a file from a floppy disk required jumping into a terminal window, logging in as administrator, and running an arcane "mount" command.

{ The author did this one the hard way. The mtools suite (open-source, naturally, and included in most Linux distributions) makes this easy. }

Networking

A very illustrative case of how the Linux user community works was revealed by my experiences with the networking subsystem.

Caldera's OpenLinux installer only provided the client daemon to handle the BootP protocol (as opposed to DHCP) and for some reasons, it didn't install correctly. I looked around on the CD that Caldera provided for a DHCP daemon and couldn't find one.

A small number of web sites and FAQs later, I found an FTP site with a Linux DHCP client. The DHCP client was developed by an engineer employed by Fore Systems (as evidenced by his email address; I believe, however, that it was developed in his own free time). A second set of documentation/manuals was written for the DHCP client by a hacker in *Hungary* which provided relatively simple instructions on how to install/load the client.

I downloaded & uncompressed the client and typed two simple commands:

Make - compiles the client binaries

Make Install -installed the binaries as a Linux Daemon

Typing "DHCPCD" (for DHCP Client Daemon) on the command line triggered the DHCP discovery process and voila, I had IP networking running.

DHCP as an example of Linux process

Since I had just downloaded the DHCP client code, on am impulse I played around a bit. Although the client wasn't as extensible as the DHCP client we are shipping in NT5 (for example, it won't query for arbitrary options & store results), it was obvious how I could write the additional code to implement this functionality. The full client consisted of about 2600 lines of code.

One example of esoteric, extended functionality that was clearly patched in by a third party was a set of routines to that would pad the DHCP request with host-speciifc strings required by Cable Modem / ADSL sites.

A few other steps were required to configure the DHCP client to auto-start and auto-configure my Ethernet interface on boot but these were documented in the client code and in the DHCP documentation from the Hungarian developer.

Key takeaways here:

- Contrary to popular belief, even though this was open source, I never had to touch the `C' code to get the core functionality working.
- The author of the driver and the author of the documentation were two geographically separated individuals.
- The GPL + incremental improvements process had *already* been at work as evidenced by the Cable Modem/ADSL extensions.
- **Most importantly:** a process that NatBro pointed out in the OSS paper - "a modestly skilled UNIX programmer can grow into doing great things with Linux". I'm a poorly skilled UNIX programmer but it was immediately obvious to me how to incrementally extend the DHCP client code (the *feeling* was exhilarating and addictive).

{ Wow. Seduced by the light side of the Force. I bet they had to send him to Microsoft re-education camp for a few weeks after this to get his indoctrination back in place... }

Additionally, due directly to GPL + having the full development environment in front of me, I was in a position where I could write up my changes and email them out within a couple of hours (in contrast to how things like this would get done in NT). Engaging in that process would have prepared me for a larger, more ambitious Linux project in the future.

Apps

Caldera bundled StarOffice from Star Corp in Germany. The Office team is quite familiar with StarOffice as a "second-string" contender in the suite category after Corel (which is bundled with Red Hat) and Lotus.

{ I wonder how familiar they really are, given that the author got the vendor's name wrong -- it's not ``Star Corp.'' but from ``Star Division Corp.''. }

StarOffice was almost entirely an Office 97 clone from a UI perspective. The menus, buttons, placement, etc. were all generally identical. In many cases, large areas of functionality in the menu bar were missing (e.g. Macros). Other stereotypical Office97 features (e.g. red squiggles under misspelled words) were correctly replicated.

As a test, I tried importing a somewhat simple PowerPoint document into StarOffice from a floppy disk. This required jumping into an x-terminal and mounting a new floppy disk into the Linux file system namespace and pointing out to Linux that it was FAT16 formatted. From there, I launched StarOffice's PowerPoint clone and pointed it at the namespace for the floppy and uploaded the file.

Simple slides (such as pure text + bullet points) imported nearly 100% correctly (although fonts and sizing were changed). Complex sides (using PowerPoint's line art, etc.) were almost always totally trashed.

{ The author of course wrote too soon to see all the excitement since August. There are likely to be no fewer than four viable, full-featured office suites in the near future (ApplixWare, Star Office, Corel/WP, and Lotus SmartSuite). A couple of these are being given away free on Linux for personal use.

And now every major database except Microsoft's is either already shipping or about to be. Database servers running on Beowulf should also be quite a shocker to Microsoft when it happens (maybe to big UNIX iron too). }

Perceived Performance

Caldera also bundles Netscape's Navigator browser. The browsers UI, of course, perfectly matches Netscape's UI on win32 platforms.

I didn't have the time to run true performance tests, but my anecdotal / percieved performance was impressive. I previously had IE4/NT4 on the same box and by comparison the combination of Linux / Navigator ran at least 30-40% faster when rendering simple HTML + graphics.

Testing end user applications on top of Linux will be an interesting performance test in the future.

On a negative note, after I had instantiated 3 instances of Navigator on the box, performance came to an almost complete standstill, the mouse become unresponsive, none of the keyboard command sequences worked and I had to reboot the box.

{ One wonders if he tried Ctrl-Alt-F[1-9], or something similar, to get out of X and back to a console login. Or asked someone to rlogin or telnet to his new box and kill his X server. Neither option would require him to reboot his machine, **that's just the only thing he knows how to do**. }

Conclusions

Skilled users with modest developer backgrounds are probably delighted to use Linux due to the endless customizability afforded by Open Source. The simplicity and

consistency of the process to modify the system presents a very low learning curve towards "joining" the Linux process.

Long term, my simple experiments do indicate that Linux has a chance at the desktop market but only after massive investments in ease of use and configuration. The average desktop user is unfamiliar with "make".

Linux Competitive Issues

Consumers Love It.

A December 1997 survey of Fortune 1000 IT shops by Datapro asked IT managers to rate their server OS's on the basis of: TCO, Interoperability, Price, Manageability, Flexibility, Availability, Java Support, Functionality, and Performance. RedHat provides summary info at: http://www.redhat.com/redhat/datapro.html.

When overall satisfaction with the OS's was calculated, Linux came out in first place. Linux was rated #1 in 7 of 9 categories in the DataPro study losing only on: functionality breadth, and performance (where it placed #2 after DEC)

Linux vs. NT

Windows NT is target #1 for the Linux community. To characterize their animosity towards NT (or, for that matter, anything Microsoft) as religious would be an understatement. Linux's (real and perceived) virtues over Windows NT include:

- **Customization** - The endless customizability of Linux for specific tasks - ranging from GFLOP clustered workstations to 500K RAM installations to dedicated, in-the-closet 486-based DNS servers - makes Linux a very natural choice for "isolated, single-task" servers such as DNS, File, Mail, Web, etc. Strict application and *OS* componentization coupled with readily exposed internals make Linux ideal.

The threat here is even more pronounced as over time, the number of servers (and consequently dedication to specific tasks) will increase. Customers enjoy the simpler debugging and fault isolation of individual servers vs a monolithic server runing multiple services.

- **Availability/Reliability** - There are hundreds of stories on the web of Linux installations that have been in continuous production for over a year. Stability more than almost any other feature is the #1 goal of the Linux development community (and the #1 cited weakness of Windows)

- **Scaleability/Performance** - Linux is considered faster than NT in networking, and processes. In particular, as a server, Linux's modular architecture allows the administrator to turn off graphics, and other non-related subsystems for extreme performance in a particular service

- **Interoperability**- Every open protocol on the planet (and many of the closed ones) have been ported to Linux. In a Windows environment, work from the SAMBA team enables Linux to look like an NT Domain Controller / File Server.

Recently, the NT performance team ran their NetBench file/print test against a recent Linux distribution. Results indicate that although NT slightly outperforms Linux, Linux's performance is still quite acceptable and competitive considering the years of tuning that has been applied to the NT SMB stack.

Linux vs. Java

Linux developers are generally wary of Sun's Java. Most of the skepticism towards Java stems directly from Sun's tight control over the language - and lack of OSS.

The Linux community has been asking Sun to treat the Linux platform as a tier-1 Java platform almost since the dawn of the language. However, Sun does NOT support the JDK for Linux.

{ This is about to change. }

Interestingly, in order to develop the Linux JDK, several Linux developers signed NDA's to develop the port (http://www.blackdown.org). These pressures have also spawned several OSS JVM clones including http://www.kaffe.org.

Linus comments in (http://www.linuxresources.com/news/linux-expo.html)

While Linus would like to see an officially supported Java Development Kit from Sun, he is still not impressed with Java and would prefer to stay out of the Microsoft/Sun clash over Java purity;

Linux vs. SunOS/Solaris

The Linux community has ambivalent feelings towards Sun. On the one hand, as the most vocal critic of Microsoft, Sun is praised. On the other hand, as the most visible yardstick in the UNIX world, beating Solaris / SunOs is a favorite past-time of Linux hackers.

Using the Lmbench OS benchmark, Linux outperforms SunOS not only on x86 but, impressively, *on Sun Hardware as well* in networking, process / context switch times, disk I/O, etc.

Some (not very scientific or comprehensive) OS performance statistics can be found on: http://www.caip.rutgers.edu/~davem/scoreboard.html.

In generating these performance results, the great number of eyes (and consequently large amount of hand tuning / optimizing of critical code paths) is most frequently cited.

A general architectural comparison citing the performance benefits of Linux over SunOS can be found on:
http://www.nuclecu.unam.mx/~miguel/uselinux/SparcLinux.html

Sun has recently announced (8/10) the free licensing of Solaris binaries for non-commercial institutions (http://www.sun.com/edu/solaris/index.html). Presumably this is due to competitive pressures from Linux.

Linux on the Server

The vast majority of Linux's installed, production base is projected to be in servers.

Reasons why Linux is strong in this market include:

- **Unix heritage** - the server market, especially at the high-end, is already familiar & comfortable with UNIX, Internet-based freeware, etc.
- **Professional users** - high end server administrators are often developers/power users themselves and are therefore comfortable with recompiling apps, etc.
- **"Generic" services** - these are services defined via open, lowest-common-denominator protocols such as DNS, SMTP, etc. Functional differentiation is lower in the server market than it is in the client market. There is a lower bar for experimentation with servers since it disrupts downstream client activity very little.
- **Dedicated Functionality** - because servers are typically tasked with a single function (e.g. mail, file/print, database, etc.), the level of required integration with other services and devices in the organization is much lower.

Network infrastructure

Linux is often used to provide commodity, low horsepower, high reliability network infrastructure services. For example:

- DNS
- DHCP
- Print Servers
- File Servers

ISP Adoption

One of Linux's core user bases is ISP shops. Some of the reasons for this include:

- **Cost** - ISP's live on horribly tiny margins. Linux's free price + wide hardware support is consequently very attractive.
- **Maintainability** - If something breaks, it needs to be fixed immediately. In larger ISPs, the technical expertise to debug code breaks or at least install quickly available patches is plentiful. Remote manageability in particular is a key attribute.
- **Reliability** - perception that non-Linux OS's aren't reliable or scalable enough (in particular Windows NT)
- **UNIX background** - ISPs are traditionally Unix havens. ISP sys admins are very well versed in arcane UNIX command line admin, remote administration, etc. In a group that's very predisposed towards UNIX's strengths, Linux represents a best-of-breed UNIX.

Thin Servers

Linux is emerging as a key operating system in the nascent thin server market:

- **Source code availability** - Freely available source code provides for easy customization of the OS
- **Commodity protocols** - Thin servers speak very simple, non-extensible, commodity protocols to clients such as HTTP, SMTP, and SMB.
- **Modularity & Small size** - Because the OS was designed in a very non-integrated, componentized manner from the outset, it's very easy to build boxes that don't have a monitor, keyboard, etc.
- **Cost** - Obviously, margins are very low in embedded devices & a free OS helps
- **Code Maintenance** - Because the Linux source is constantly being upgraded, embedded developers are reassured that new changes / fixes can be snapped back to their systems at any time.
- **Tool Availability** -Unix tools are far more powerful than the current crop of embedded development tools.

One of the most prominent thin-server on the market based on Linux today is the Cobalt Microserver (http://www.cobaltmicro.com). Other thin server vendors (most notably Whistle Interjet) are using FreeBSD derived products.

Case Study: Cisco Systems, Inc.

IDC published a study of 3 corporate IS departments which had significantly deployed Linux. Cisco has several hundred Linux servers deployed through their organization serving the following functions:

- NFS/SMB server
- Print Server (LPD & SMB)
- Small office productivity (ApplixWare office suite, Netscape Navigator)
- WWW Server & Proxy
- Software development

Linux on the Client

Due to it's UNIX heritage and Hacker OS background, Linux is a weak client-desktop OS. Additionally, the OSS paper points out why, in a broad sense, OSS is much more of a server threat than a desktop threat.

There are, however, several initiatives attempting to push Linux as a viable desktop replacement. Each of the various Desktop environments (GNOME, KDE, CDE) come bundled with basic productivity applications and there are 2 full fledged office suite products (from Corel and StarOffice) which provide varying degrees of file format compatibility with Microsoft Office.

App / GUI Chaos

Unlike the Kernel - where Linus Torvalds maintains the core source tree, the Linux GUI has NOT been singularly managed and consequently has a highly forked tree.

Linux does not have a consistent UI look and feel due to the variety of widget sets (a widget is analogous to an OCX or VBX) such as Motif, LessTiff, MIT Athena, Sun OpenLook, etc. Because widgets represent central UI concepts (such as a close button, dropdown menu, dialog box, etc.), users get different look-and-feels and often different usage semantics.

In addition to Widgets, the "desktop" or "shell" has also forked. Primary players in the shell arena include:

- **Common Desktop Environment (CDE) --** a collaboration between major commercial Unix manufacturers. CDE, however, is not GPL'd and has thus resulted in multiple Linux groups creating CDE replacements. CDE is available on Linux.

- **K Desktop Environment (KDE) -** a "free" CDE clone. KDE replaces all functionality in CDE but does not provide a widget set. (in practice the widget set is actually MORE lines of code than the desktop). Consequently, the KDE developers chose the QT widget set which was most liberally licensed - but still not GPL'd - and compatible with most other UNIX systems. This however, launched the final band of GPL zealots who created...

- **Gnome** - a radical new UI initiative based loosely on X-windows and incorporating CORBA into the desktop. While this is an ambitious task, and may be more revolutionary than CDE is, its long from being complete and is lacking in application support.

The lack of singular, customer-focused management has resulted in the unwillingness to compromise between the different initiatives and is evident of the management costs in the Linux process.

Unix Developers

Linux as a client has found a home with UNIX developers. Many developers prefer to use Linux for their dev machines in order to write code for other UNIX platforms. The ease of debugging on top of a platform where there is open source is often cited.

Non-PC Devices

Corel's NC devices were based on a Linux derived OS. These efforts, however, have since been suspended (with the Corel developed application-level code being returned to the OSS community)

Linux Forecasts & Futures

Current Initiatives / Linux Futures

There are literally hundreds of small research projects attempting to improve various parts of the Linux OS.

Some projects include:

- **Linux 2.2** - High Availability features such as deeper RAID support (RAID 0, 1, 5 supported today), volume management; file system performance improvements; asynchronous I/O & completion ports; Ipv6; . An excellent feature summary can be found on: http://lwn.net/980730/a/2.2chFinal.html.

- **Linux 3.0** -- Linus forecasts that the next version of the Kernel will incorporate better SMP scalability and begin to attack the clustering problem. Development is far from starting so details / commitments are extremely sketchy.

- **Beowulf clustering** - Beowulf is a shared-nothing cluster that runs today on Linux. It requires specially developed applications which are able to spawn subprocesses on remote hosts for computing. As such, it is not a real competitor to WolfPack and most of the magic in Beowulf in the applications rather than system services. However, as a press-magnet, Beowulf clusters with appropriate software have been demonstrated at supercomputer power (a 10GFLOP was recently ranked #315 on the top 500 supercomputers list maintained by the NCSA).

- **DIPC** - Distributed Inter-Process Control Pack - provides standard IPC functions to client apps (semaphores, shared memory, etc.) but is able to remote those functions to network hosts.

- **GNOME** -- Next generation UI initiative for Linux loosely based on X-windows +CORBA . More info at http://www.gnome.org. Many of the key developers for Gnome work for RedHat.

"Parity Growth"

The biggest future issue for Linux is what to do once they've reached parity with UNIX. JimAll used the phrase "chasing taillights" to captures the core issue: in the fog of the market place, you can move faster by being "number 2 gaining on number 1" than by being number 1.

Linux has now reached parity / incrementally ahead of other Unixes. Consequently, it will be much harder to achieve the big leaps the development team is accustomed to.

From Wired's piece on Linux:

This two-track development process has made Linux probably more advanced and yet more stable than any other version of Unix today. "Linux is now entering an era of pure development instead of just catching up," says Jacques Gélinas.

Strengths

A second paper on "Open Source Software" goes into depth on the generic advantages of the Open Source Process.

Unix Heritage & Fast Copying

Linux unabashedly steals the best ideas from the various UNIX flavors. This means free R&D. Recently, Linux has begun to copy NT-ish features such as transmitfile(), a hacked form of IO Completion Ports, etc.

Established / high-visibility bazaar

Linux is the most often cited example of a "credible" open source project. By being the largest OSS project today, it's the most sustainable in the future.

Dominance In Education / Research Markets

New ideas from academia + new computer scientists are being trained wholesale in the Linux OS. In particular, Europe and Asia are very hooked on the Linux OS. Email from BartelB (Marketing Manager EdCU):

For higher education in particular, Linux represents an alternative to the Commercial demons of software, (not a quantitative statement but in talking with many CS students who supply 60% of the labor for higher education IT departments, they have express these feelings and its a problem). They feel that once they commit to a windows platform there creativity will be lost. Money is not there driving force, they don't want to be "Borged".

Weaknesses

The paper on "Open Source Software" provides general process weaknesses. Here, we'll try to list only the weaknesses that are unique to Linux.

Unix Heritage

Linux's biggest advantage can also quickly become a disadvantage - particularly in volume markets where ease of use is paramount. Some nascent efforts have been launched to make Linux friendlier but they are generally receiving relative apathy from the dev community (http://www.seul.org).

Too Many Managers

In a typical Linux distribution, the majority of the code comes from sources outside of the main Linux tree. This piecemeal approach will make it especially hard to solve architectural problems and launch new, cross-component initiatives.

Worst case scenarios

This section is pure speculation. What are some of the worst case scenarios for Linux to hurt Microsoft?

Customer Adoption - It gets good enough

Using today's server requirements, Linux is a credible alternative to commercial developed servers in many, high volume applications. The effect of this on our server revenue model would be immense.

Our client-side revenue model is still strong however for a variety of reasons including switching costs for the entire pool of win32 source code. Linux advocates, however, are working on various emulators and function call impersonators to attack this cost.

This points back to an obvious solution - innovation in the core platform is an ongoing requirement.

Channel Adoption

The "Open Source Software" paper has a section on OSS business models. Summarizing that section, there are 4 primary business models we have identified for Open Source Software.

1. **Secondary Services** - The vendor / developer of OSS makes their money on service contracts, customer integration, etc.
2. **Loss Leader for Market Entry** -- The vendor / developer of OSS uses OSS's process advantages (in particular credibility) as a lever against established commercial vendors.
3. **Commoditizing Downstream Suppliers** -- The vendor / developer of OSS is also the producer of a product / service further in the value chain and closer to the consumer.
4. **Standards Preemption** - Because OSS process are argued to be winner-take-all, it may suit the vendor / developer to seed the OSS market with their codebase to pre-empt a competitive codebase from taking hold.

IBM Adopts Linux?

IBM is most capable at capturing revenues from all 4 of the business models associated with Linux.

1. **Secondary Services** - IBM is very strong in consulting, integration, support, etc. This is their fastest growing business today
2. **Loss Leader** - IBM's client/low-end operating system business is in shambles (remember OS/2?). Additionally, IBM has stumbled on various NC/JavaOS systems as well. By leveraging Linux's credibility (as well as applying IBM's development resources toward improving ease of use?), IBM would hope to upset the status quo in the volume OS space and hope to capture revenue in the ensuing disruption.
3. **Commoditizing Downstream Suppliers** - As a PC/Hardware OEM, IBM's margins increase by commoditizing a key cost item -- the OS. In particular, the commoditized & highly customizable qualities of the Linux OS actually provide greater differentiation for hardware vendors.
4. **Standards Preemption** - The standard to pre-empt is anything Microsoft - in particular new OS services that we integrate directly into future versions of NT.

IBM, despite their Apache announcements, seems unlikely to advocate this in the short run. I'd imagine that religion within their various OS development efforts alone would provide a significant amount of near term inertia.

Sun Adopts?

Sun's rationale for adopting Linux would be less encompassing than IBM's.

1. **Secondary Services** - Sun is not very strong in consulting / integration revenue. They do, however, make significant revneue in support and maintenance.
2. **Loss Leader** – Sun could market Linux as a low-end OS and try to make money in the UNIX applications space above it. Because Linux could potentially be a far larger market than anything Sun is accustomed to, this would be a net positive for them.
3. **Commoditizing Downstream Suppliers** - Sun is also a hardware vendor (with some excellent systems). Sun would lose their current OS revenue but the ability to sell their hardware into a broader channel could be compelling.

Linux adoption, however, puts Sun at significant risk if their SPARC operations cannot keep up with Intel's innovation pace.

4. **Standards Preemption** - Beat Microsoft Standards

PC OEM's

Other worst case adoption scenarios are subsets of the Sun / IBM case and involve other PC vendors such as Compaq and Dell.

Note, however, that Compaq and Dell merely have to credibly threaten Linux adoption in order to push for lower OEM OS pricing.

{ Ohhh, that's a **good** one! Hit 'em in the cash flow, guys! }

Server ISVs

One interesting spin on the "Commoditizing Downstream Suppliers" strategy could be **backward integration** by server ISV's. For example, Oracle could ship a version of Parallel Server for Linux that *includes the Linux OS within the distribution.*

This is basically a play on the thin-server concept. Instead of *integrating* multiple small business functions on a single server, this attempts to *disintegrate* the features of an enterprise OS into the minimal set necessary to run the specific server application. It plays into the business models identified as follows:

1. **Secondary Services -** Companies like Oracle/SAP/Baan/etc. already make a large percentage of their income from on-site consulting agreements

2. **Loss Leader** - treating the OS as a loss leader helps them concentrate revenues for a particular hardware unit into their hands

3. **Downstream commoditization** - Oracle has no problem declaring the Server OS as a kernel, memory manager, IP stack, and some disk.

4. **Standards Preemption -** beat Microsoft.

Next Steps & Microsoft Response

A lot more thought and work needs to go into formulating Microsoft's response to Linux. Some initial thoughts on how to compete with Linux in particular are contained below. One "blue sky" avenue that should be investigated is if there is any way to turn Linux into an opportunity for Microsoft.

A more generalized assessment of how to beat the Open Source Software process which begat Linux is contained in the "Open Source Software" document.

Beating Linux

Beat UNIX

The single biggest contributor to Linux's success is the general viability of the UNIX market. Systematically attacking UNIX in general helps attack Linux in particular. Some Linux-targeted initiatives in this space (not a comprehensive list) include:

- **Improve Low-End "IAM"** – Scaleability, Interoperability, Availability, and Management (SIAM) are the most often cited reasons for using UNIX over NT in mission critical, high-end applications.

In today's Linux deployments however, scaleability is not the driver as much as Interop, Reliabiliity, and Headless Management.

- **UNIX services for NT Add-on pack**

Modularize / Embed Windows NT

Relative to other UNIX's Linux is considered more *customizable*. Addressing this functionality involves more than just the embedded Windows NT project. Greater componentatization & general dependency reduction within NT will improve not only it's stability but also the ability of highly skilled users/admins to deploy task-specific NT installations.

This requires:

- Wide availability of the Embedded NT toolkit
- Greater focus on ease-of-use in the toolkit

Beat commodity protocols / services

Linux's homebase is currently commodity network and server infrastructure. By folding extended functionality into today's commodity services and create new protocols, we raise the bar & change the rules of the game.

Some of the specifics mentioned in the OSS paper:

- **DNS integration with Directory**. Leveraging the Directory Service to add value to DNS via dynamic updates, security, authentication

- **HTTP-DAV**. DAV is complex and the protocol spec provides an infinite level of implementation complexity for various applications (e.g. the design for Exchange over DAV is good but certainly not the single obvious design). Apache will be hard pressed to pick and choose the correct first areas of DAV to implement.

- **Structured storage**. Changes the rules of the game in the file serving space (a key Linux/Apache application). Create a compelling client-side advantage which can be extended to the server as well (e.g. heterogenous join of client & server datastores).

- **MSMQ for Distributed Applications**. MSMQ is a great example of a distributed technology where most of the value is in the services and implementation and NOT in the wire protocol.

Leverage ISV's for system improvements

A key long term advantage that Linux will enjoy is the massive pool of developers willing to improve areas of the core platform. Microsoft will never be able to employ a similar headcount.

A key mechanism to combat this is to make it easy (and provide incentives) for ISV's to extend system components in NT for custom, vertical applications. One example here could be Veritas' specialized file system drivers for NT.

"WinTone"

Linux's modularity and customization also implies inconsistencies in services available on an arbitrary Linux installation. Microsoft can provide a bundle of services that are universally available in all OS releases (current initiatives include WBEM-based management) that generate network externalities when combined across many devices in the network.

Put another way, the extreme modularity of Linux devalues what a "Linux-logo'ed" app means. By contrast, Window's monolithic nature gives an app developer more leeway in terms of what API's are callable.

Process Vulnerabilities

Where is Microsoft vulnerable to Linux? As stated earlier, the primary threat resides on the server vs. the client.

Linux will "Cream Skim" the Best NT Server Features

The Linux community is very willing to copy features from other OS's if it will serve their needs. Consequently, there is the very real long term threat that as MS expends the development dollars to create a bevy of new features in NT, Linux will simply cherry pick the best features an incorporate them into their codebase.

The effect of patents and copyright in combatting Linux remains to be investigated.

Linux is recreating the MS "3rd release is a charm" advantage - FASTER

Microsoft's market power doesn't stem from products as much as it does from our iterative process. The first release of a Microsoft product often fairs poorly in the market and primarily of generates fine granularity feedback from consumers. Similarly, Linux has shown that they are capable of iterative cycles - but at an order of magnitude faster rate. On the flip side, however, our incremental releases are arguably much larger whereas many of Linux's incremental releases are tantamount to pure bug fixing.

{ Perish forbid that anyone should ever issue a release just to **fix bugs**. }

The GNU Manifesto

GNU, which stands for Gnu's Not Unix, is the name for the complete Unix-compatible software system which I am writing so that I can give it away free to everyone who can use it. (1) Several other volunteers are helping me. Contributions of time, money, programs and equipment are greatly needed.

So far we have an Emacs text editor with Lisp for writing editor commands, a source level debugger, a yacc-compatible parser generator, a linker, and around 35 utilities. A shell (command interpreter) is nearly completed. A new portable optimizing C compiler has compiled itself and may be released this year. An initial kernel exists but many more features are needed to emulate Unix. When the kernel and compiler are finished, it will be possible to distribute a GNU system suitable for program development. We will use TeX as our text formatter, but an nroff is being worked on. We will use the free, portable X window system as well. After this we will add a portable Common Lisp, an Empire game, a spreadsheet, and hundreds of other things, plus on-line documentation. We hope to supply, eventually, everything useful that normally comes with a Unix system, and more.

GNU will be able to run Unix programs, but will not be identical to Unix. We will make all improvements that are convenient, based on our experience with other operating systems. In particular, we plan to have longer file names, file version numbers, a crashproof file system, file name completion perhaps, terminal-independent display support, and perhaps eventually a Lisp-based window system through which several Lisp programs and ordinary Unix programs can share a screen. Both C and Lisp will be available as system programming languages. We will try to support UUCP, MIT Chaosnet, and Internet protocols for communication.

GNU is aimed initially at m achines in the 68000/16000 class with virtual memory, because they are the easiest machines to make it run on. The extra effort to make it run on smaller machines will be left to someone who wants to use it on them.

To avoid horrible confusion, please pronounce the `G' in the word `GNU' when it is the name of this project.

Why I Must Write GNU

I consider that the golden rule requires that if I like a program I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. For years I worked within the Artificial Intelligence Lab to resist such tendencies and other inhospitalities, but eventually they had gone too far: I could not remain in an institution where such things are done for me against my will.

So that I can continue to use computers without dishonor, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free. I have resigned from the AI lab to deny MIT any legal excuse to prevent me from giving GNU away.

Why GNU Will Be Compatible with Unix

Unix is not my ideal system, but it is not too bad. The essential features of Unix seem to be good ones, and I think I can fill in what Unix lacks without spoiling them. And a system compatible with Unix would be convenient for many other people to adopt.

How GNU Will Be Available

GNU is not in the public domain. Everyone will be permitted to modify and redistribute GNU, but no distributor will be allowed to restrict its further redistribution. That is to say, proprietary (18k characters) modifications will not be allowed. I want to make sure that all versions of GNU remain free.

Why Many Other Programmers Want to Help

I have found many other programmers who are excited about GNU and want to help.

Many programmers are unhappy about the commercialization of system software. It may enable them to make more money, but it requires them to feel in conflict with other programmers in general rather than feel as comrades. The fundamental act of friendship among programmers is the sharing of programs; marketing arrangements now typically used essentially forbid programmers to treat others as friends. The purchaser of software must choose between friendship and obeying the law. Naturally, many decide that friendship is more important. But those who believe in law often do not feel at ease with either choice. They become cynical and think that programming is just a way of making money.

By working on and using GNU rather than proprietary programs, we can be hospitable to everyone and obey the law. In addition, GNU serves as an example to inspire and a banner to rally others to join us in sharing. This can give us a feeling of harmony which is impossible if we use software that is not free. For about half the programmers I talk to, this is an important happiness that money cannot replace.

How You Can Contribute

I am asking computer manufacturers for donations of machines and money. I'm asking individuals for donations of programs and work.

One consequence you can expect if you donate machines is that GNU will run on them at an early date. The machines should be complete, ready to use systems, approved for use in a residential area, and not in need of sophisticated cooling or power.

I have found very many programmers eager to contribute part-time work for GNU. For most projects, such part-time distributed work would be very hard to coordinate; the independently-written parts would not work together. But for the particular task of replacing Unix, this problem is absent. A complete Unix system contains hundreds of utility programs, each of which is documented separately. Most interface specifications are fixed by Unix compatibility. If each contributor can write a compatible replacement for a single Unix utility, and make it work properly in place of the original on a Unix system, then these utilities will work right when put together. Even allowing for Murphy to create a few unexpected problems, assembling these components will be a feasible task. (The kernel will require closer communication and will be worked on by a small, tight group.)

If I get donations of money, I may be able to hire a few people full or part time. The salary won't be high by programmers' standards, but I'm looking for people for whom building community spirit is as important as making money. I view this as a way of enabling dedicated people to devote their full energies

to working on GNU by sparing them the need to make a living in another way.

Why All Computer Users Will Benefit

Once GNU is written, everyone will be able to obtain good system software free, just like air.(2)

This means much more than just saving everyone the price of a Unix license. It means that much wasteful duplication of system programming effort will be avoided. This effort can go instead into advancing the state of the art.

Complete system sources will be available to everyone. As a result, a user who needs changes in the system will always be free to make them himself, or hire any available programmer or company to make them for him. Users will no longer be at the mercy of one programmer or company which owns the sources and is in sole position to make changes.

Schools will be able to provide a much more educational environment by encouraging all students to study and improve the system code. Harvard's computer lab used to have the policy that no program could be installed on the system if its sources were not on public display, and upheld it by actually refusing to install certain programs. I was very much inspired by this.

Finally, the overhead of considering who owns the system software and what one is or is not entitled to do with it will be lifted.

Arrangements to make people pay for using a program, including licensing of copies, always incur a tremendous cost to society through the cumbersome mechanisms necessary to figure out how much (that is, which programs) a person must pay for. And only a police state can force everyone to obey them. Consider a space station where air must be manufactured at great cost: charging each breather per liter of air may be fair, but wearing the metered gas mask all day and all night is intolerable even if everyone can afford to pay the air bill. And the TV cameras everywhere to see if you ever take the mask off are outrageous. It's better to support the air plant with a head tax and chuck the masks.

Copying all or parts of a program is as natural to a programmer as breathing, and as productive. It ought to be as free.

Some Easily Rebutted Objections to GNU's Goals

"Nobody will use it if it is free, because that means they can't rely on any support."

"You have to charge for the program to pay for providing the support."

If people would rather pay for GNU plus service than get GNU free without service, a company to provide just service to people who have obtained GNU free ought to be profitable.(3)

We must distinguish between support in the form of real programming work and mere handholding. The former is something one cannot rely on from a software vendor. If your problem is not shared by enough people, the vendor will tell you to get lost.

If your business needs to be able to rely on support, the only way is to have all the necessary sources and tools. Then you can hire any available person to fix your problem; you are not at the mercy of any individual. With Unix, the price of sources puts this out of consideration for most businesses. With GNU this will be easy. It is still possible for there to be no available competent person, but this problem cannot be blamed on distribution arrangements. GNU does not eliminate all the world's problems, only some of them.

Meanwhile, the users who know nothing about computers need handholding: doing things for them which they could easily do themselves but don't know how.

Such services could be provided by companies that sell just hand-holding and repair service. If it is true that users would rather spend money and get a product with service, they will also be willing to buy the service having got the product free. The service companies will compete in quality and price; users will not be tied to any particular one. Meanwhile, those of us who don't need the service should be able to use the program without paying for the service.

"You cannot reach many people without advertising, and you must charge for the program to support that."

"It's no use advertising a program people can get free."

There are various forms of free or very cheap publicity that can be used to inform numbers of computer users about something like GNU. But it may be true that one can reach more microcomputer users with advertising. If this is really so, a business which advertises the service of copying and mailing GNU for a fee ought to be successful enough to pay for its advertising and more. This way, only the users who benefit from the advertising pay for it.

On the other hand, if many people get GNU from their friends, and such companies don't succeed, this will show that advertising was not really necessary to spread GNU. Why is it that free market advocates don't want to let the free market decide this?(4)

"My company needs a proprietary operating system to get a competitive edge."

GNU will remove operating system software from the realm of competition. You will not be able to get an edge in this area, but neither will your competitors be able to get an edge over you. You and they will compete in other areas, while benefiting mutually in this one. If your business is selling an operating system, you will not like GNU, but that's tough on you. If your business is something else, GNU can save you from being pushed into the expensive business of selling operating systems.

I would like to see GNU development supported by gifts from many manufacturers and users, reducing the cost to each.(5)

"Don't programmers deserve a reward for their creativity?"

If anything deserves a reward, it is social contribution. Creativity can be a social contribution, but only in so far as society is free to use the results. If programmers deserve to be rewarded for creating innovative programs, by the same token they deserve to be punished if they restrict the use of these programs.

"Shouldn't a programmer be able to ask for a reward for his creativity?"

There is nothing wrong with wanting pay for work, or seeking to maximize one's income, as long as one does not use means that are destructive. But the means customary in the field of software today are based on destruction.

Extracting money from users of a program by restricting their use of it is destructive because the restrictions reduce the amount and the ways that the program can be used. This reduces the amount of wealth that humanity derives from the program. When there is a deliberate choice to restrict, the harmful consequences are deliberate destruction.

The reason a good citizen does not use such destructive means to become wealthier is that, if everyone did so, we would all become poorer from the mutual destructiveness. This is Kantian ethics; or, the Golden Rule. Since I do not like the consequences that result if everyone hoards information, I am required to consider it wrong for one to do so. Specifically, the desire to be rewarded for one's creativity does not justify depriving the world in general of all or part of that creativity.

"Won't programmers starve?"

I could answer that nobody is forced to be a programmer. Most of us cannot manage to get any money for standing on the street and making faces. But we are not, as a result, condemned to spend our lives standing on the street making faces, and starving. We do something else.

But that is the wrong answer because it accepts the questioner's implicit assumption: that without ownership of software, programmers cannot possibly be paid a cent. Supposedly it is all or nothing.

The real reason programmers will not starve is that it will still be possible for them to get paid for programming; just not paid as much as now.

Restricting copying is not the only basis for business in software. It is the most common basis because it brings in the most money. If it were prohibited, or rejected by the customer, software business would move to other bases of organization which are now used less often. There are always numerous ways to organize any kind of business.

Probably programming will not be as lucrative on the new basis as it is now. But that is not an argument against the change. It is not considered an injustice that sales clerks make the salaries that they now do. If programmers made the same, that would not be an injustice either. (In practice they would still make considerably more than that.)

"Don't people have a right to control how their creativity is used?"

"Control over the use of one's ideas" really constitutes control over other people's lives; and it is usually used to make their lives more difficult.

People who have studied the issue of intellectual property rights carefully (such as lawyers) say that there is no intrinsic right to intellectual property. The kinds of supposed intellectual property rights that the government recognizes were created by specific acts of legislation for specific purposes.

For example, the patent system was established to encourage inventors to disclose the details of their inventions. Its purpose was to help society rather than to help inventors. At the time, the life span of 17 years for a patent was short compared with the rate of advance of the state of the art. Since patents are an issue only among manufacturers, for whom the cost and effort of a license agreement are small compared with setting up production, the patents often do not do much harm. They do not obstruct most individuals who use patented products.

The idea of copyright did not exist in ancient times, when authors frequently copied other authors at length in works of non-fiction. This practice was useful, and is the only way many authors' works have survived even in part. The copyright system was created expressly for the purpose of encouraging authorship. In the domain for which it was invented--books, which could be copied economically only on a printing press--it did little harm, and did not obstruct most of the individuals who read the books.

All intellectual property rights are just licenses granted by society because it was thought, rightly or wrongly, that society as a whole would benefit by granting them. But in any particular situation, we have to ask: are we really better off granting such license? What kind of act are we licensing a person to do?

The case of programs today is very different from that of books a hundred years ago. The fact that the easiest way to copy a program is from one neighbor to another, the fact that a program has both source code and object code which are distinct, and the fact that a program is used rather than read and enjoyed, combine to create a situation in which a person who enforces a copyright is harming society as a whole both materially and spiritually; in which a person should not do so regardless of whether the law enables him to.

"Competition makes things get done better."

The paradigm of competition is a race: by rewarding the winner, we encourage everyone to run faster. When capitalism really works this way, it does a good job; but its defenders are wrong in assuming it always works this way. If the runners forget why the reward is offered and become intent on winning, no matter how, they may find other strategies--such as, attacking other runners. If the runners get into a fist fight, they will all finish late.

Proprietary and secret software is the moral equivalent of runners in a fist fight. Sad to say, the only referee we've got does not seem to object to fights; he just regulates them ("For every ten yards you run, you can fire one shot"). He really ought to break them up, and penalize runners for even trying to fight.

"Won't everyone stop programming without a monetary incentive?"

Actually, many people will program with absolutely no monetary incentive. Programming has an irresistible fascination for some people, usually the people who are best at it. There is no shortage of professional musicians who keep at it even though they have no hope of making a living that way.

But really this question, though commonly asked, is not appropriate to the situation. Pay for programmers will not disappear, only become less. So the right question is, will anyone program with a reduced monetary incentive? My experience shows that they will.

For more than ten years, many of the world's best programmers worked at the Artificial Intelligence Lab for far less money than they could have had anywhere else. They got many kinds of non-monetary rewards: fame and appreciation, for example. And creativity is also fun, a reward in itself.

Then most of them left when offered a chance to do the same interesting work for a lot of money.

What the facts show is that people will program for reasons other than riches; but if given a chance to make a lot of money as well, they will come to expect and demand it. Low-paying organizations do poorly in competition with high-paying ones, but they do not have to do badly if the high-paying ones are banned.

"We need the programmers desperately. If they demand that we stop helping our neighbors, we have to obey."

You're never so desperate that you have to obey this sort of demand. Remember: millions for defense, but not a cent for tribute!

"Programmers need to make a living somehow."

In the short run, this is true. However, there are plenty of ways that programmers could make a living without selling the right to use a program. This way is customary now because it brings programmers and businessmen the most money, not because it is the only way to make a living. It is easy to find other ways if you want to find them. Here are a number of examples.

A manufacturer introducing a new computer will pay for the porting of operating systems onto the new hardware.

The sale of teaching, hand-holding and maintenance services could also employ programmers.

People with new ideas could distribute programs as freeware, asking for donations from satisfied users, or selling hand-holding services. I have met people who are already working this way successfully.

Users with related needs can form users' groups, and pay dues. A group would contract with programming companies to write programs that the group's members would like to use.

All sorts of development can be funded with a Software Tax:

Suppose everyone who buys a computer has to pay x percent of the price as a software tax. The government gives this to an agency like the NSF to spend on software development.

But if the computer buyer makes a donation to software development himself, he can take a credit against the tax. He can donate to the project of his own choosing--often, chosen because he hopes to use the results when it is done. He can take a credit for any amount of donation up to the total tax he had to pay.

The total tax rate could be decided by a vote of the payers of the tax, weighted according to the amount they will be taxed on.

The consequences:

- The computer-using community supports software development.
- This community decides what level of support is needed.
- Users who care which projects their share is spent on can choose this for themselves.

In the long run, making programs free is a step toward the post-scarcity world, where nobody will have to work very hard just to make a living. People will be free to devote themselves to activities that are fun, such as programming, after spending the necessary ten hours a week on required tasks such as legislation, family counseling, robot repair and asteroid prospecting. There will be no need to be able to make a living from programming.

We have already greatly reduced the amount of work that the whole society must do for its actual productivity, but only a little of this has translated itself into leisure for workers because much nonproductive activity is required to accompany productive activity. The main causes of this are bureaucracy and isometric struggles against competition. Free software will greatly reduce these drains in the area of software production. We must do this, in order for technical gains in productivity to translate into less work for us.

Footnotes

(1) The wording here was careless. The intention was that nobody would have to pay for *permission* to use the GNU system. But the words don't make this clear, and people often interpret them as saying that copies of GNU should always be distributed at little or no charge. That was never the intent; later on, the manifesto mentions the possibility of companies providing the service of distribution for a profit. Subsequently I have learned to distinguish carefully between "free" in the sense of freedom and "free" in the sense of price. Free software is software that users have the freedom to distribute and change. Some users may obtain copies at no charge, while others pay to obtain copies--and if the funds help support improving the software, so much the better. The important thing is that everyone who has a copy has the freedom to cooperate with others in using it.

(2) This is another place I failed to distinguish carefully between the two different meanings of "free". The statement as it stands is not false--you can get copies of GNU software at no charge, from your friends or over the net. But it does suggest the wrong idea.

(3) Several such companies now exist.

(4) The Free Software Foundation raises most of its funds from a distribution service, although it is a charity rather than a company. If *no one* chooses to obtain copies by ordering them from the FSF, it will be unable to do its work. But this does not mean that proprietary restrictions are justified to force every user to pay. If a small fraction of all the users order copies from the FSF, that is sufficient to keep the FSF afloat. So we ask users to choose to support us in this way. Have you done your part?

(5) A group of computer companies recently pooled funds to support maintenance of the GNU C Compiler.

GNU General Public License

Version 2, June 1991

59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

Terms and conditions for copying, distribution and modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire

whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free

status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No Warranty

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

one line to give the program's name and an idea of what it does.
Copyright (C) *yyyy name of author*

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA. Also add information on how to contact you by electronic and paper mail. If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) *year name of author*

Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'. This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989

Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

The Cluetrain Manifesto

1. Markets are conversations.
2. Markets consist of human beings, not demographic sectors.
3. Conversations among human beings sound human. They are conducted in a human voice.
4. Whether delivering information, opinions, perspectives, dissenting arguments or humorous asides, the human voice is typically open, natural, uncontrived.
5. People recognize each other as such from the sound of this voice.
6. The Internet is enabling conversations among human beings that were simply not possible in the era of mass media.
7. Hyperlinks subvert hierarchy.
8. In both internetworked markets and among intranetworked employees, people are speaking to each other in a powerful new way.
9. These networked conversations are enabling powerful new forms of social organization and knowledge exchange to emerge.
10. As a result, markets are getting smarter, more informed, more organized. Participation in a networked market changes people fundamentally.
11. People in networked markets have figured out that they get far better information and support from one another than from vendors. So much for corporate rhetoric about adding value to commoditized products.
12. There are no secrets. The networked market knows more than companies do about their own products. And whether the news is good or bad, they tell everyone.
13. What's happening to markets is also happening among employees. A metaphysical construct called "The Company" is the only thing standing between the two.
14. Corporations do not speak in the same voice as these new networked conversations. To their intended online audiences, companies sound hollow, flat, literally inhuman.
15. In just a few more years, the current homogenized "voice" of business—the sound of mission statements and brochures—will seem as contrived and artificial as the language of the 18th century French court.
16. Already, companies that speak in the language of the pitch, the dog-and-pony show, are no longer speaking to anyone.
17. Companies that assume online markets are the same markets that used to watch their ads on television are kidding themselves.
18. Companies that don't realize their markets are now networked person-to-person, getting smarter as a result and deeply joined in conversation are missing their best opportunity.
19. Companies can now communicate with their markets directly. If they blow it, it could be their last chance.
20. Companies need to realize their markets are often laughing. At them.
21. Companies need to lighten up and take themselves less seriously. They need to get a sense of humor.
22. Getting a sense of humor does not mean putting some jokes on the corporate web site. Rather, it requires big values, a little humility, straight talk, and a genuine point of view.
23. Companies attempting to "position" themselves need to take a position. Optimally, it should relate to something their market actually cares about.
24. Bombastic boasts—"We are positioned to become the preeminent provider of XYZ"—do not constitute a position.
25. Companies need to come down from their Ivory Towers and talk to the people with whom they hope to create relationships.
26. Public Relations does not relate to the public. Companies are deeply afraid of their markets.
27. By speaking in language that is distant, uninviting, arrogant, they build walls to keep markets at bay.
28. Most marketing programs are based on the fear that the market might see what's really going on inside the company.
29. Elvis said it best: "We can't go on together with suspicious minds."
30. Brand loyalty is the corporate version of going steady, but the breakup is inevitable—and coming fast. Because they are networked, smart markets are able to renegotiate relationships with blinding speed.
31. Networked markets can change suppliers overnight. Networked knowledge workers can change employers over lunch. Your own "downsizing initiatives" taught us to ask the question: "Loyalty? What's that?"
32. Smart markets will find suppliers who speak their own language.
33. Learning to speak with a human voice is not a parlor trick. It can't be "picked up" at some tony conference.
34. To speak with a human voice, companies must share the concerns of their communities.
35. But first, they must belong to a community.
36. Companies must ask themselves where their corporate cultures end.
37. If their cultures end before the community begins, they will have no market.
38. Human communities are based on discourse—on human speech about human concerns.
39. The community of discourse is the market.
40. Companies that do not belong to a community of discourse will die.
41. Companies make a religion of security, but this is largely a red herring. Most are protecting less against competitors than against their own market and workforce.
42. As with networked markets, people are also talking to each other directly inside the company—and not just about rules and regulations, boardroom directives, bottom lines.
43. Such conversations are taking place today on corporate intranets. But only when the conditions are right.

44. Companies typically install intranets top-down to distribute HR policies and other corporate information that workers are doing their best to ignore.
45. Intranets naturally tend to route around boredom. The best are built bottom-up by engaged individuals cooperating to construct something far more valuable: an intranetworked corporate conversation.
46. A healthy intranet organizes workers in many meanings of the word. Its effect is more radical than the agenda of any union.
47. While this scares companies witless, they also depend heavily on open intranets to generate and share critical knowledge. They need to resist the urge to "improve" or control these networked conversations.
48. When corporate intranets are not constrained by fear and legalistic rules, the type of conversation they encourage sounds remarkably like the conversation of the networked marketplace.
49. Org charts worked in an older economy where plans could be fully understood from atop steep management pyramids and detailed work orders could be handed down from on high.
50. Today, the org chart is hyperlinked, not hierarchical. Respect for hands-on knowledge wins over respect for abstract authority.
51. Command-and-control management styles both derive from and reinforce bureaucracy, power tripping and an overall culture of paranoia.
52. Paranoia kills conversation. That's its point. But lack of open conversation kills companies.
53. There are two conversations going on. One inside the company. One with the market.
54. In most cases, neither conversation is going very well. Almost invariably, the cause of failure can be traced to obsolete notions of command and control.
55. As policy, these notions are poisonous. As tools, they are broken. Command and control are met with hostility by intranetworked knowledge workers and generate distrust in internetworked markets.
56. These two conversations want to talk to each other. They are speaking the same language. They recognize each other's voices.
57. Smart companies will get out of the way and help the inevitable to happen sooner.
58. If willingness to get out of the way is taken as a measure of IQ, then very few companies have yet wised up.
59. However subliminally at the moment, millions of people now online perceive companies as little more than quaint legal fictions that are actively preventing these conversations from intersecting.
60. This is suicidal. Markets want to talk to companies.
61. Sadly, the part of the company a networked market wants to talk to is usually hidden behind a smokescreen of hucksterism, of language that rings false—and often is.
62. Markets do not want to talk to flacks and hucksters. They want to participate in the conversations going on behind the corporate firewall.
63. De-cloaking, getting personal: We are those markets. We want to talk to you.
64. We want access to your corporate information, to your plans and strategies, your best thinking, your genuine knowledge. We will not settle for the 4-color brochure, for web sites chock-a-block with eye candy but lacking any substance.
65. We're also the workers who make your companies go. We want to talk to customers directly in our own voices, not in platitudes written into a script.
66. As markets, as workers, both of us are sick to death of getting our information by remote control. Why do we need faceless annual reports and third-hand market research studies to introduce us to each other?
67. As markets, as workers, we wonder why you're not listening. You seem to be speaking a different language.
68. The inflated self-important jargon you sling around—in the press, at your conferences—what's that got to do with us?
69. Maybe you're impressing your investors. Maybe you're impressing Wall Street. You're not impressing us.
70. If you don't impress us, your investors are going to take a bath. Don't they understand this? If they did, they wouldn't let you talk that way.
71. Your tired notions of "the market" make our eyes glaze over. We don't recognize ourselves in your projections—perhaps because we know we're already elsewhere.
72. We like this new marketplace much better. In fact, we are creating it.
73. You're invited, but it's our world. Take your shoes off at the door. If you want to barter with us, get down off that camel!
74. We are immune to advertising. Just forget it.
75. If you want us to talk to you, tell us something. Make it something interesting for a change.
76. We've got some ideas for you too: some new tools we need, some better service. Stuff we'd be willing to pay for. Got a minute?
77. You're too busy "doing business" to answer our email? Oh gosh, sorry, gee, we'll come back later. Maybe.
78. You want us to pay? We want you to pay attention.
79. We want you to drop your trip, come out of your neurotic self-involvement, join the party.
80. Don't worry, you can still make money. That is, as long as it's not the only thing on your mind.
81. Have you noticed that, in itself, money is kind of one-dimensional and boring? What else can we talk about?
82. Your product broke. Why? We'd like to ask the guy who made it. Your corporate strategy makes no sense. We'd like to have a chat with your CEO. What do you mean she's not in?
83. We want you to take 50 million of us as seriously as you take one reporter from The Wall Street Journal.

84. We know some people from your company. They're pretty cool online. Do you have any more like that you're hiding? Can they come out and play?
85. When we have questions we turn to each other for answers. If you didn't have such a tight rein on "your people" maybe they'd be among the people we'd turn to.
86. When we're not busy being your "target market," many of us are your people. We'd rather be talking to friends online than watching the clock. That would get your name around better than your entire million dollar web site. But you tell us speaking to the market is Marketing's job.
87. We'd like it if you got what's going on here. That'd be real nice. But it would be a big mistake to think we're holding our breath.
88. We have better things to do than worry about whether you'll change in time to get our business. Business is only a part of our lives. It seems to be all of yours. Think about it: who needs whom?
89. We have real power and we know it. If you don't quite see the light, some other outfit will come along that's more attentive, more interesting, more fun to play with.
90. Even at its worst, our newfound conversation is more interesting than most trade shows, more entertaining than any TV sitcom, and certainly more true-to-life than the corporate web sites we've been seeing.
91. Our allegiance is to ourselves—our friends, our new allies and acquaintances, even our sparring partners. Companies that have no part in this world, also have no future.
92. Companies are spending billions of dollars on Y2K. Why can't they hear this market timebomb ticking? The stakes are even higher.
93. We're both inside companies and outside them. The boundaries that separate our conversations look like the Berlin Wall today, but they're really just an annoyance. We know they're coming down. We're going to work from both sides to take them down.
94. To traditional corporations, networked conversations may appear confused, may sound confusing. But we are organizing faster than they are. We have better tools, more new ideas, no rules to slow us down.
95. We are waking up and linking to each other. We are watching. But we are not waiting.

Free Software Leaders Stand Together

The Craig Mundie speech is old news by now, so hopefully this is the last word. A number of the free software evangelists, in informal discussion, felt that the proper response to Microsoft would be to stand together. Mundie's speech shows that Microsoft's strategy is to keep us divided and attack us one at a time, until all are gone. Thus, their emphasis on the GPL this time. While we didn't try to represent every group and project, many major voices of Open Source and Free Software have signed this message. We took a while, because we're not used to this, but we'll be better next time. So, please note the signatures at the bottom of this message - we will stand together, and defend each other.

-Bruce Perens

We note a new triumph for Open Source and Free Software: we have become so serious a competitor to Microsoft that their executives publicly announce their fear. However, the only threat that we present to Microsoft is the end of monopoly practices. Microsoft is welcome to participate as an equal partner, a role held today by entities ranging from individuals to transnational corporations like IBM and HP. Equality, however, isn't what Microsoft is looking for. Thus, they have announced Shared Source, a system that could be summarized as Look but don't touch - and we control everything.

Microsoft deceptively compares Open Source to failed dot-com business models. Perhaps they misunderstand the term Free Software. Remember that Free refers to liberty, not price. The dot-coms gave away goods and services as loss-leaders, in unsuccessful efforts to build their market share. In contrast, the business model of Open Source is to reduce the cost of software development and maintenance by distributing it among many collaborators.

The success of the Open Source model arises from copyright holders relaxing their control in exchange for more and better collaboration. Developers allow their software to be freely redistributed and modified, asking only for the same privileges in return.

There is much software that is essential to a business, but which does not differentiate that business from its competitors. Even companies that have not fully embraced the Open Source model can justify collaboration on Free Software projects for this non-differentiating software, because of the money they will save. And such collaborations are often overwhelmingly successful: for example, the project that produces the market-leading Apache web server was started by a group of users who agreed to share the work of maintaining a piece of software that each of their businesses depended on.

The efficiency of this cooperation is in the best interests of the user. But Free Software is also directly in the user's interest, because it means that the users control the software they use. When they do business with Open Source vendors, the vendors do not dominate them.

With very little funding, the GNU/Linux system has become a significant player in many major markets, from Internet servers to embedded devices. Our GUI desktop projects have astounded the software industry by going from zero to being comparable with or superior to others in only 4 years. Workstation manufacturers like Sun and HP have selected our desktops to replace their own consortium projects, because our work was better. An entire industry has been built around Free Software, and is growing rapidly despite an unfavorable market. The success of software companies like Red Hat, and the benefits to vendors such as Dell and IBM, demonstrate that Free Software is not at all incompatible with business.

The Free Software license singled out for abuse by Microsoft is the GNU General Public License, or GNU GPL. This license is the computer equivalent of share and share alike. But this does not mean, as Microsoft claims, that a company using these programs is legally obliged to make all its software and data free. We make all GPL software available in source form for incorporation as a building block in new programs. This is the secret of how we have been able to create so much good software, so quickly.

If you do choose to incorporate GPL code into a program, you will be required to make the entire program Free Software. This is a fair exchange of our code for yours, and one that will continue as you reap the benefit of improvements contributed by the community. However, the legal requirements of the GPL apply only to programs which incorporate some of the GPL-covered code - not to other programs on the same system, and not to the data files that the programs operate upon.

Although Microsoft raises the issue of GPL violations, that is a classic red herring. Many more people find themselves in violation of Microsoft licenses, because Microsoft doesn't allow copying, modification, and redistribution as the GPL does. Microsoft license violations have resulted in civil suits and imprisonment. Accidental GPL violations are easily remedied, and rarely get to court.

It's the share and share alike feature of the GPL that intimidates Microsoft, because it defeats their Embrace and Extend strategy. Microsoft tries to retain control of the market by taking the result of open projects and standards, and adding incompatible Microsoft-only features in closed-source. Adding an incompatible feature to a server, for example, then requires a similarly-incompatible client, which forces users to "upgrade". Microsoft uses this deliberate-incompatibility strategy to force its way through the marketplace. But if Microsoft were to attempt to "embrace and extend" GPL software, they would be required to make each incompatible "enhancement" public and available to its competitors. Thus, the GPL threatens the strategy that Microsoft uses to maintain its monopoly.

Microsoft claims that Free Software fosters incompatible "code forking", but Microsoft is the real motor of incompatibility: they deliberately make new versions incompatible with old ones, to force users to purchase each upgrade. How many times have users had to upgrade Office because the Word file format changed? Microsoft claims that our software is insecure, but security experts say you shouldn't trust anything but Free Software for critical security functions. It is Microsoft's programs that are known for snooping on users, vulnerability to viruses, and the possibility of hidden "back doors".

Microsoft's Shared Source program recognizes that there are many benefits to the openness, community involvement, and innovation of the Open Source model. But the most important component of that model, the one that makes all of the others work, is freedom. By attacking the one license that is specifically designed to fend off their customer and developer lock-in strategy, they hope to get the benefits of Free Software without sharing those benefits with those who participate in creating them.

We urge Microsoft to go the rest of the way in embracing the Open Source software development paradigm. Stop asking for one-way sharing, and accept the responsibility to share and share alike that comes with the benefits of Open Source. Acknowledge that it is compatible with business.

Free Software is a great way to build a common foundation of software that encourages innovation and fair competition. Microsoft, it's time for you to join us.

Bruce Perens, Primary Author: The Open Source Definition

co-signers:

Richard Stallman, Free Software Foundation.
Eric Raymond, Open Source Initiative.
Linus Torvalds, Creator of the Linux Kernel.
Miguel de Icaza, GNOME GUI Desktop Project.
Larry Wall, Creator of the Perl Language.
Guido van Rossum, Creator of the Python Language.
Tim O'Reilly, Publisher.
Bob Young, Co-Founder, Red Hat
Larry Augustin, CEO, VA Linux Systems

Suggested Web Sites

The following links present more information regarding open source:

Open Source Philosophy

www.opensource.org
The Open Source Initiative. Non-profit leaded by Eric Raymond. During the last 4 years has done a lot regarding the public relations of the Open Source Movement.

www.fsf.org
The Free Software Foundation. Non-profit leaded by Richard Stallman. The originators of the GPL and original Open Source and Free software movement.

www.oreilly.com/catalog/opensources/book/toc.html
An online (and also printed) book containing essays regarding Open Source.

Open Source News and Community Sites

www.lwn.net
Linux Weekly News. An excellent news site.

www.slashdot.org
Excellent community site. A good portion of the debate is centred on Linux and Open Source.

www.linuxjournal.com
The Linux Journal. An authoritative voice Linux and Open Source.

Open Source Advocacy

www.dwheeler.com/oss_fs_why.html
Very comprehensive compilation of up to date facts regarding Open Source market share, performance, reliability, and other data.

pel.cs.byu.edu/~alen/computers/Linux/WhyLinux/Why.html
Why Linux, a collection of Linux “propaganda”.

www.datasync.com/~rogerspl/Advocacy-HOWTO.html#toc4
Linux Advocacy Mini-HOWTO.

Open Source Software

www.freshmeat.net
Place to search for Open Source Software.

www.sourceforge.net
The largest Open Source development site and collaboration platform.

linas.org/linux/
Linux Enterprise Computing. A good list of enterprise software projects for Linux.

Index