A Practice-Based Model of Access for Science:
Linux Kernel Development and Shared Digital Resources

Matt Ratto

In this paper I argue that analyses of access to the contexts and work of 'e-science' and scientific 'cyberinfrastructures' are hindered by models that assume fixed roles for contributors and users and undervalue the joint 'reworking' of scientific data that is one of the central strengths of such approaches. Using a community of Free/Libre Open Source software (FLOSS) developers as a complementary case, I develop an alternative practice-based model of access that focuses on the particular sets of social and technical knowledges that allow individuals to work together to develop and maintain shared resources. Importantly, this model of access puts the practices of 'reworking' as central rather than peripheral to human activity. Access within this framework is characterized as the ability to shift between individual and joint, mediated work, and to understand and manipulate the multiple representations of shared objects such shifts require.

Keywords: open access, data sharing, distributed work, Linux, Free/open source software

Though most scholars would agree that arguments about the internet as essentially socially liberating have been grossly over-stated, digital networks have encouraged new possibilities for the coordination and consolidation of shared work. These possibilities are increasingly being adopted in scientific and scholarly fields, with many disciplines creating new digital infrastructures to facilitate the distribution, sharing, and archiving of scientific data and other forms of scholarly information. More than just electronic storage facilities, these 'e-science' networks and databases are often predicated on 'quid-pro-quo' relationships; since access to the stored information requires participation, scholars and scientists must give information to get information. Examples of scientific work that involves the use of digital objects are myriad but include shared data archives of brain imagery (e.g. Beaulieu, 2004) and biodiversity databases (e.g. Bowker, 2000; Casey, 2003). Equally, many social science and humanities archives and infrastructures are being created that rely on digital resources, including shared corpora for linguistic analysis (Fry, 2003), the creation of community libraries (Wright et al., 2002) and other initiatives such as those being developed under the auspices of new e-social science humanities funding, such as the UK's ESCR National Centre for e-Social Science; the Netherlands' Virtual Knowledge Studio, and the US' Commission on Cyberinfrastructure for the Humanities and Social Sciences. An important marker in the shift towards more distributed forms of science and scholarship was the creation in 2003 of the Office of Cyberinfrastructure by the US National Science Foundation.

This ongoing move to more distributed forms of scientific and scholarly work has raised a number of questions by practitioners, funding agencies, infrastructure developers, and scholars, who all share an interest in creating systems of knowledge production along more open, collaborative, and decentralized lines. These questions include how to make these infrastructures as open as possible without sacrificing the quality of the information they contain, how to support the diversity of research needs and also develop tools for specific requirements, and how to support traditional research while simultaneously helping scientists and scholars pose novel questions and new forms of enquiry. Needless to say, these issues all involve, to a greater or lesser degree, negotiations of access to the spaces and the objects of distributed work.

In this paper I construct a practice-based model of access that explores the specific means by which individuals negotiate the complex social and technical landscape involved in digitally-mediated work, claiming that such a model can help scientists, scholars, and infrastructure designers create social and technical systems that foster inclusion, manage issues of quality, and maintain the specificity of the scholarly or scientific objects particular to their discipline. While previous models of access to scientific data such as those summarized below can adequately address questions of access within bounded and highly structured institutional frameworks, new distributed scientific practices that rely on the Internet and digital data resemble much more the messy, often disorganized work involved in free/libre/open source (FLOSS) software. This being the case, a study of how FLOSS developers manage questions of access can help us understand the questions raised by highly distributed and digitally-mediated scholarship.

While the argument has often been made that FLOSS development in some ways resembles Mertonian science (e.g. Kelty, 2001), I argue instead that science and scholarship, in its moves towards increasingly distributed formations, in some ways resembles FLOSS. Therefore, I examine the shared work practices of a group of developers responsible for the maintenance and ongoing development of a key element of the Linux operating system.[1] Linux is considered one of the primary successes of the FLOSS software engineering method, characterized by a distributed form of development, carried out mostly on the Internet, by a group of geographically distributed volunteers. A key element in the development of Linux is the Linux Kernel Mailing List (LKML), a shared space/place where contributors to Linux can exchange source code and debate issues related to coding practice, the organizational structure of Linux development, as well as possible future directions for the community.

The paper consists of the following sections; in section I, I provide an example of some current definitions of access and, through a brief discussion of Linux development, demonstrate how they are inadequate for describing how participation within the Linux community, and correspondingly, distributed scholarship and science, is mediated. In section II, I lay out a practice-based model of access through an extended discussion of the shifts in practices and objects that characterize the ongoing work of Linux development. In section III I visually represent the ideas of reworking articulated in section II, using the resultant chart to detail how successful access in Linux development progresses. Finally, in section IV, I reflect on how this notion of access can help practitioners, funding agencies, infrastructure designers, and scholars better understand the specific needs of scientists and scholars working in distributed contexts, concluding with some final insights about the relationship between digitality and the objects and representations of science.

Data and methods

The following analysis of access in Linux kernel development results from a five year qualitative ethnographic and historiographic study on Linux development conducted primarily between 1998-2003. The data sources used in that study included the web pages of Linux developers, online and offline journalism about Linux development, analyses of Linux source code itself, and observation and participation on email lists related to Linux development. These are reported in more depth in Ratto (2003; 2005a; 2005b). The primary source used in the present article is the Linux Kernel Mailing List (LKML) archive hosted at Indiana University.[2] This online archive contains emails from as far back as 1996, and, as of May 1, 2003 contained over 1.6 gigabytes of individual entries. In 2002-2003 with the permission of the archive personnel I copied approximately 1 GB of this material to a local Unix-based computer in order to carry out more complex searches than were possible with the online system. The majority of quotes reproduced in the sections below are a result of searches on a partial sample of the overall LKML archive using the *grep* search tool. The key search terms were generated through a *grounded theory* analysis (Glaser and Strauss, 1967; Glaser, 1992; Strauss and Corbin, 1990; 1997) which followed the following overlapping phases; first, ongoing observation and conversations about development activity; second, collection and reading of relevant materials; third, an iterative process of coding and memoing gathered materials; fourth, the development of categories and properties that help explain aspects of development; fifth, an emergent process of writing and theory building that describes relations between these categories and properties. The quotes used in section IV of this paper come from a follow-up study on the relationship between FLOSS and scientific practice carried out in 2006-2007.

I. Defining access, openness, and Gnu/Linux

The role of ‘open access’ in current science policy (e.g. Houghton et al., 2003; NIH, 2003; Goldenberg-Hart, 2004) demonstrates the importance of conceptualizing access as an active, ongoing practice. This was highlighted in an article by the OECD Follow-up Group on Issues of Access to Publicly Funded Research

Data (Arzberger et al., 2004). In this article the authors critiqued the recent focus on increasing access to the output of scientific work, (i.e. scientific publications), rather than addressing what they saw as a more serious problem: the lack of access to the raw material of scientific work, namely research data. Access to research data often requires a 'quid-pro-quo' between research scientists, in that it is not merely about passive access to static material, but is predicated instead on a relationship to shared material and the common construction of a joint resource. Understanding this relationship as requiring an ongoing process of negotiation within a system of join, expert work requires rethinking traditional definitions of access.

Access

Access, in relationship to issues involving the use of technologies or scientific knowledge, is typically conceptualized as the ability or right to obtain, make use of, or take advantage of something.[3] This concept of access is typically analyzed by mapping a set of properties in order to characterize relationships between providers and consumers of data or services. For example, in their analysis of the health care industry, Penchansky and Thomas (1981) focused on 'the 5 A's', affordability, availability, accessibility, accommodation, and acceptability. Other analyses of access have focused on properties such as transparency and information management, (e.g. Schenkelaars and Ahmad, 2004) and/or the need for incentives and reward structures to overcome resistance. (e.g. Houghton et al., 2003).

Similarly, in the article by the OECD Follow-up Group (Arzberger et al., 2004) mentioned above, the researchers divided issues of access into five domains; Institutional & Managerial, Legal & Policy, Financial & Budgetary, Cultural & Behavioral, and Technological, all linked in reciprocal relationships. (Arzberger et al., 2004: 144.) An important addition to previous descriptions of access, this model includes cultural and behavioral factors as an essential category of properties to address when considering questions related to access, rather than relegating it to a subordinate position. Under this category, the OECD group placed properties related to trust in research quality, codes of conduct based in professionalism, and how flexible members of a research community might be in approaching permutations of their rules and procedures. Another important aspect of this model is that it demonstrates the connected nature of the multiple domains of access, e.g. that financial issues are connected to legal context which is in turn influenced by technological considerations.

Such mappings of access are demonstrably useful as one way to evaluate and/or problematize current systems of access to important social resources. By directing focus towards the various aspects of access, these categories can provide a framework for examining what kinds of restrictions may manifest themselves, and what domains (legal, economic, cultural, etc.) may be involved. However, this framework requires both a clear definition of the boundaries between the various aspects (e.g. what counts as 'institutional and managerial' and what can be considered 'legal and policy'), as well as static moments in which access can be analyzed. Thus while it might work very well for highly structured and bounded institutional contexts where both responsibilities and contexts of access are clear and can be described, this framework is not as useful for less structured contexts, such as those of distributed science and FLOSS work. Further, this model does not differentiate between the work that is shared – research data is considered to be the same whatever particular context of work it relates to, be it anthropology, biology, geology, or computer science. This is particularly problematic for cyberinfrastructure and e-science, given the ways in which such infrastructures are expected to move beyond the passive database to become contexts for shared work in which scientific objects are actively transformed, reworked, and repurposed. What previous models of access do not adequately address is the role scientific objects themselves play in how systems can and should be designed.

Given the complexity of the interconnecting issues, a more dynamic and practice-based model of access will better serve those interested in diagnosing and designing shared digital resources. FLOSS development and the practices of Linux Kernel developers in particular, provide a rich resource for exploring questions of access.

Gnu/Linux

Gnu/Linux is an operating system, like Microsoft Windows or Apple's Mac OS that provides services (such as interfaces, file systems, and application environments) to personal computer users. Gnu/Linux is currently running on approximately 29 – 35 % of all computer servers, as well as on a much smaller percentage of desktop computers. Some estimates put the current number of total Gnu/Linux users at around 18 million.[4]

Unlike Windows or MacOS, Gnu/Linux has been mostly developed by a geographically distributed group of volunteer developers, who have used the Internet and a license called the General Public License, or GPL, to maintain access to the results of their work. 'Linux' although often used as a short-hand for the total operating system, more properly names the effort to create what is probably the most complex part of this system, its kernel.[5, 6] The originator of Linux, Linus Torvalds, along with a loose and changing group of developers has maintained ongoing development since 1991, making it one of the longest projects of this type. The result of this project is a set of programs, containing more than 15 million lines of computer code.[7] Thus, Linux demonstrates that 'open source' efforts in developing large-scale software projects can be successful, despite traditional theories about software development. (e.g. Brooks, 1975.)

While early journalism about Linux often reported its success as a direct result of the abilities of the instigator of the project, Linus Torvalds, academic work has tended to stress a number of additional pre-conditions such as an existing group of computer experts, knowledgeable about the Unix software environment and resistant to efforts by Unix owners to control access to the source code (Raymond, 2001; 2003), accessible free/open source tools and the internet (Moon and Sproul, 2000) and a 'hacker ethic' discourse that valued open and ongoing development (Himanen, 2001; Torvalds, 2001). In general, work on FLOSS has primarily addressed issues of reputation, trust, and value (e.g. Pavlicek, 2000; Kelty, 2001), the importance of the Internet in the coordination of developer activity (e.g. Kollock and Smith, 1999; Preece, 2000), and the relationship of FLOSS to traditional forms of development, economics, and markets (e.g. Ghosh, 1998, Benkler, 2002). However, more recently, an ethnographic focus has begun to emerge, looking in more detail at the specific historical and cultural practices involved in simultaneously creating code and coding communities (Hakken, 1999; Coleman, 2005; Kelty, 2005).

Pressure of openness

In my own ethnographic work on Linux (Ratto, 2003; 2005a; 2005b) I pursue the idea that its success is due to the ways Linux developers manage a problem of access that is shared by all large-scale, distributed open source development efforts. This problem emerges from the need to balance openness with coordination: on one hand staying open to new directions of software development, the incorporation of new ideas, and new members of the development community, but on the other hand, to remain focused on continuing development, organizing effort, and maintaining direction. Together these two, often contradictory needs create what I term the 'pressure of openness', a pressure faced by all infrastructures of shared, distributed work. The success of Linux is in no small part due to its productive managing of this pressure through deliberate and often accidental manipulation of the following community characteristics: context, membership, social organization, and shared objects and modes of communication. In the following sections I provide some brief contextualization of each category, noting in particular where similar characteristics can also be found in the practices, objects, and organization of science. The sections below thus serve both to explicate some of the specificities of FLOSS as well as its similarity to science organizations and practices.

Context of work

The Linux kernel community is made possible by the Linux Kernel Mailing List (LKML) conversation which, along with the kernel source code itself, is one of its most visible products. The LKML provides rich data for examining the way the kernel designers create the kernel software as well as create the culture which sustains this development. There the people who are working most closely with the development of Linux discuss with one another their problems, strategies, dreams, and arguments. This conversation includes email posts about the structure of the community as well as the artifact, and often includes pieces of the Linux kernel itself – the source code. Thus, the LKML represents more than 'just talk'—retrospective traces of the work of Linux kernel developers—and is instead a context that, along with the kernel software code itself, organizes and structures the practices of kernel development. Understanding online discussions as dynamic spaces of shared work rather than as passive traces of past process is important for understanding both FLOSS and digitally distributed science.

Performing membership

Rather than designing for 'users', many Linux developers seem to be creating software programs and libraries inherently directed towards other programmers with the skills and knowledge to re-work the programs for their own use. One important similarity between Linux developers and other communities of workers oriented around the construction and use of shared digital resources is the (often unintentional) inward focus of development. Free/Libre/Open Source Software (FL/OSS) projects (and Linux among them) are often criticized for their lack of attention to usability issues (e.g. Nickell, 2001; Eklund et al., 2002; Nichols and Twidale, 2003).[8] Rather than develop software for a wide and diverse audience of users, many open source developers create projects directed towards users like themselves – technically sophisticated and computer-savvy individuals from similar backgrounds. This blending of user and developer often results in software that, while technically sophisticated, remains difficult to use and problematic to learn for non-computer professionals. This is equally the case when it comes to participating within the Linux development community. While designing for re-use does create communities that blend making and using, the skills and knowledge required to participate in this community puts membership out of reach for many computer users.The high barrier to participating is not accidental. Rather, it provides an important policing function for the community as a whole. The need for self-selection was clearly made in the initial post announcing the Linux project:

> Do you pine for the nice days of Minix-1.1, when men were men and wrote their own device drivers? Are you without a nice project and just dying to cut your teeth on a OS you can try to modify for your needs? Are you finding it frustrating when everything works on Minix? No more all-nighters to get a nifty program working? Then this post might be just for you :-)[9]

Note the nostalgia for an earlier day in computing history when all users 'wrote their own device drivers' (and were men). Projects, modification, all-nighters – these types of activities and the knowledge required to partake in them, can be understood as characterizing Linux work. Although it can be rightly argued that since the early days of development, Linux has become a more 'user-centric' system, the number of times the quote reproduced above is called forth by Linux developers (and social researchers) indicates the continuing importance of this characterization. As I explore later in this article, the continuing adherence to these traditions demonstrate how important practices of reworking and re-use are to members.

Social organization

While it is important to note the success of Linux as a free and open source software development effort, it is also important not to over-emphasize the 'openness' of this community. The LKML is not a moderated list, meaning that anyone with an email account and internet access can post to it. However, this does not necessarily mean that anyone can contribute to the Linux kernel, or that contributions will be included in one of the official source code trees. Dave Jones, one of the main developers, graphically represented the relationship between developers working concurrently on the Linux kernel:

> The closest approximation my minds-eye can make of how things work look something like this..

```
h h h h h
\ | | | /
 m m m
  \ |/
  ttt
   |
   l
```

> h – random j hacker working on same file/subsystem different goals
> m – maintainer for file/subsys
> t – 'forked' tree maintainer (-ac, -dj, -aa etc..)
> l – Linus
>
> Whilst development happens concurrently in parallel, the notion of progress is somewhat serialized as changes work their way down to Linus.
> (This whole thing goes a little astray when random j hacker sends patches straight to Linus bypassing everyone else and they get merged, but the controlled anarchy prevails and everyone somehow gets back in sync). [10]

Note the hierarchical organizational structure, with Linus Torvalds at the bottom and 'random j hackers' at the top. Like in science, the community is structurally organized with number of past contributions, time served, and overall participation playing some role in the coordination of hierarchy and the acceptance of new contributions by the community.

Shared objects

While the increasing use of source management systems has had an impact on Linux kernel development, a key activity of Linux development is predicated on the idea of the 'patch'. Source management systems provide more complete modes of control over the entire source code 'tree' than previous 'patch and tarball' modes of management. This is useful for systems like the Linux kernel that contain many multiple code subsystems and many million lines of code. However, despite the adoption of first Bitkeeper (2002-2005) and more recently GIT (2005-) kernel development remains reliant on patches as the smallest unit of change. Submitting and 'committing' patches, particularly for non-lead developers (those not responsible for large kernel sub-systems) is still an important part of the development process.

Linux kernel development occurs through a process by which sections of existing source code are supplemented or changed by applying 'patches'. Patch, a software utility originally created by Larry Wall and since maintained and updated by the Free Software Foundation (FSF), allows segments of a program's source code to be changed without having to completely overwrite the whole program.[11] Patch serves as an interface to another GNU utility, 'diff', a software tool that discovers differences between files. In order to create a patch file, a programmer uses diff to compare the old source code version against the one he or she has changed. Diff generates a file containing just the lines that are different between the two versions. The programmer then uses patch to create a patch file containing only the different lines. Others who want to incorporate the new changes then only have to download the patch file and use the patch utility to apply it to their existing source code.

Patching is important, particularly in an FLOSS development context, since the rule 'release early and often' tends to result, at least initially, in software requiring frequent updates in order to fix bugs.[12] Since these changes are often small, typically effecting less than 5% of the total program, patching makes the process of incremental development that is key to FLOSS possible. Rather than having to download or get physical media (such as floppy disks, tapes, or CDROM's) of the complete source code of a program in order to incorporate new changes, one applies the much smaller patch file using the patch utility. Linux kernel development is dependent upon the patch utility, since the source code files for the complete kernel are quite large.[13] Further, the LKML itself is used to discuss kernel patches as well as to distribute them – kernel patches are typically submitted to it for review. Through the LKML, the programmers active in kernel development manage their own individual and group projects, ask and give advice, discuss programming and organizational issues, and importantly, post patches that incrementally upgrade and change the Linux kernel. The LKML thus contains both commentary and code as well as a combination of both individual and social work.

Shared communication norms
Jointly creating and distributing patches and patch code requires adherence to a set of shared communication practices, articulated on the LKML, in its FAQ, and on the web pages of developers. The LKML FAQ details the 'proper' way to contribute patches to the list, including how to format the patch, the email by which it is sent, as well as who should receive it. Eric Raymond, himself both a theorist about free and open source software as well as a contributor to the Linux kernel, also has some advice:

> It is very difficult to judge the quality of code. So developers tend to evaluate patches by the quality of the submission. They look for clues in the submitter's style and communications behavior instead — indications that the person has been in their shoes and understands what it's like to have to evaluate and merge an incoming patch… experience teaches that patches which look careless or are packaged in a lazy and inconsiderate way are very likely to actually be bogus.[14]

While Raymond's advice is meant to be applicable for all FLOSS projects, his comments ring particularly true for the reworking of code on the LKML. For example, in the post below, a developer posts a new patch that adds support for the Marvell model of hard drive controller, the low level electronics that allow hard drives to work.

> This is the first public release of my libata compatible low level driver for the Marvell SATA family. Currently it successfully runs in PIO mode on a 6081 chip. EDMA support is in the works and should be done shortly. Review, testing (especially on other flavors of Marvell), comments welcome.
> (code continues…)[15]

In this post, the developer announces the public release of his patch, describes briefly what it does, and provides the source code as formatted according to LKML rules. This is immediately acknowledged by the maintainer for the effected Linux subsystem and a conversation begins as to its technical aspects. Alternatively, in a post later that same day another coder announces a patch he has created to help regulate electronics in telecommunication equipment that runs Linux.

> The following is a driver I would like to see included in the base kernel.
> It allows OS control of a device that synchronizes signaling hardware across a ATCA chassis…
> (source code continues)[16]

The response to this post is very different, with initial comments on incorrect coding style, spelling errors, and the existence of commented out (non-working) code within the patch itself. It is only after the original author addresses these issues that a technical conversation emerges about the actual workings of the patch.

Being an open and non-moderated list, anyone can post to the LKML. However, participating in Linux kernel development work requires taking on both coding and communication standards. Analogous in many ways to those involved in science communities, these standards, while often debated and transformed, provide a point of reference for both new and old participants. They include rules on how to format code, who should/should not be included in particular conversations, and how to ask for help. Adherence to the communicative codes of Linux is made more important by the 'pressure of openness' mentioned earlier. Given the openness of the community, performing membership through the modes and forms of appropriate communication serves as a more tacit but no less structural constraint on access than restricting access through online moderation or passwords.

Access to Linux

The brief overview of Linux described above provides good evidence of the complex technical, social, and communicative knowledge that must be learned in order to have what might be called 'access' to Linux kernel development activity. Linux development, like scientific and technical work more generally, is typically more than just an ongoing individualistic engagement with the rules of nature. Instead, this work is often explicitly social, including processes of inclusion, acknowledgment of past workers, and the convincing of other members of a research or engineering effort. A potential participant must be knowledgeable of the past history of Unix and Linux at least in so far as it applies to the 'hacker ethic'. They must be able to 'write their own device drivers,' in other words have the necessary subject knowledge to be able to contribute to the community by developing source code. They must be able to negotiate the social and communicative norms that define appropriate forms of participation on the LKML, the shared space of development, including knowing their place within the hierarchy of the Linux organization and knowing how to submit a patch. All of these aspects come together to form what it means to have access to Linux kernel development. Note that all of these aspects are not required to have access to the result of this work, the Linux kernel itself. However, if one wants to participate in development and to have a role in the shared work of Linux developers, then the above aspects must be interpolated into individual working practices. In other words, access to Linux kernel development involves both the right to contribute and exchange source code but also entry into the Linux kernel community. This is similar in many ways to the ways in which access is negotiated within science communities as well, where access entails acceptance or at least participation in the shared structures of scientific legitimation and institutionalization.

It is possible to place the above aspects of performed membership, shared communication and social norms, and knowledge of the shared space and object, within the framework of access described above by Arzberger et al. (2004). However, while this framework provides a valuable starting point, it does not go far enough in understanding the complex and interactive work involved in expert, digitally mediated work. A more dynamic model of access is required to address contexts marked by contributory activity, digital objects, and shared work, one that take into consideration the ways such work takes place in 'communities of practice' (Lave and Wenger, 1991) and requires participation and 'apprenticeship' (Collins, 1987).

II. Developing a practice-based model of access: understanding mediation

There is increasing recognition of the mediated nature of human creativity, that invention is not a 'de novo' internal act, but one that involves working with and in a materially and socially mediated space. For example, in *Cognition in the Wild*, Hutchins (1995) explores how cognition itself is not merely 'in the head' but consists instead of a distributed network that includes other people, representations, and artifacts. Similar insights about joint, mediated activity include those related to Distributed Collective Practices and the tradition of Activity Theory (Wertsch, 1981; Cole, 1985; Engeström, 1987).[17] Uncovering what kinds of social work is ongoing in practices of science and technology is particularly important in the context of access, especially in contexts where access involves the right to exchange artifacts and services, but also requires entry to a specific community of practioners, entry that facilitates not just the exchange of fixed objects, but also the shared, joint manipulation of them.

While positions on mediated human activity are far from monolithic (Cole, 1996: 139), there are some general similarities. First, they share an interest in the simultaneously productive and communicative nature of social behavior (Rossi-Landi, 1983). Second, they all tend to emphasize the dialetical character of human experience, seeing structure and agency as similarly determined and determining. (Lave and Chaiklin, 1993). Third, and most importantly, these theories understand cognition as 'distributed', incorporating individual human subjects, the built environment, and other people.

Such perspectives are a good starting point for examining the shared work of the Linux developers detailed above, as well as the increasingly distributed nature of scientific activity. Taken together, I suggest that most expert work involves re-working and re-purposing existing objects within shared community spaces rather than the creation of entirely novel artifacts in individual and isolated contexts.[18] Practices of 'reworking' can thus provide a starting point for understanding how access is managed in distributed, shared work. In order to better clarify the particular quality and aspects of these practices, I use two common practices: that of 'tinkering' and of 'redesigning' in order to draw a spectrum of 'reworking'. These terms have a double importance, first, because these are terms that are often used in conceptual explorations of scientific and engineering work and second, because they are used on the LKML itself as descriptions of reworking activities.

Practices of reworking: tinkering and redesigning

If science and engineering is generally a process of reworking rather than 'de novo' innovation, such is certainly the case with Linux kernel development. Scacchi (2004: 62), for example, understands 'reinvention' as an important part of FLOSS work, where '…sharing, examining, modifying, and redistributing concepts and techniques that have appeared [elsewhere],' is seen as primary. Similarly, Lin has noted the importance of 'tinkering' for ongoing work that involves ICTs and particularly FLOSS computing (Lin, 2004). Gasser et al. (2003: 2) on the other hand emphasize the need to understand how FLOSS developers handle the problems of 'redesign': 'how does a globally dispersed community of FLOSS developers design and redesign complex software systems in a continuously emergent manner?' Further, they use the term 'continuous design' to refer to the constantly updating, designing, and reusing practices associated with FLOSS development. As they note, this is a complex process that includes a lack of 'formal design processes', the sharing of 'software development informalisms', and an 'emergent knowledge process'. (Gasser et al., 2003).

What these commentators emphasize is the ways in which both tinkering and redesigning feature as important aspects of the reworking practices involved in distributed, open source work. However, while redesigning is typically understood as a conceptual, thoughtful, and ultimately professional act, tinkering is often described as unthoughtful and amateur manipulation of material resources. Opening up these definitions requires examining how these terms are used in relevant literature as well as by members of the Linux community.

Tinkering

Studies of scientific and technical work have demonstrated the importance of tinkering for science and engineering. Knorr-Cetina (1979), for example, uses 'tinkering' to refer to practices in which scientists make incremental and ad-hoc changes in the material infrastructures they use to accomplish scientific goals: 'Doable' work as tinkering is emphasized in Fujimura's article "The construction of doable problems in cancer research" (Fujimura, 1987). In addition, Norris says that "science is tinkery business" (Norris, 1993). More specifically, Nutch (1996) lays out a list of 'modes' associated with tinkering. Summarizing these modes, they include: using objects designed for other purposes, creating research equipment from bits and pieces found around the research site, modifying available tools, instruments, and equipment for coping with specific emergencies or project contingency, and saving time and money by constructing a needed piece of equipment rather than buying it through 'conventional channels'. These definitions of tinkering describe it as a cognitively rich process, one that forms a key aspect of expert work. This is equally demonstrated in anthropological analyses; in his ethnography *Working Knowledge*, Harper (1992) uses the term to explain Willie, a 'jack-of-all-trades' auto mechanic, carpenter, and metalworker in upstate New York. Harper considers Willie "...first as a thinker: considering, reconsidering, always with a view to what is available." (Harper 1992: 74)

Consequently, 'tinkering' stand out as a practice that involves a specific relationship between people and objects mediated by 'immediacy' and contingency. Tinkering is the accomplishment of 'doable' work – doable in the moment, making use of existing, rather than distant material resources. In other words, tinkering is first and foremost a practice of using immediately available resources to accomplish accessible tasks.

This perspective on tinkering can be witnessed in action on the LKML. For example, when programmers on the LKML say that they are 'tinkering' with a section of the kernel code, we can understand that they are working directly on the code itself, trying things out, adding syntax or changing algorithms in 'real-time,' quickly moving between source code and compiled code. Here are a few examples from the list itself:

> ...I was **tinkering** with zipslack this time and attempted to mount my slightly COD zip drive as umsdos. Fine. Did an ls.....oops....seg fault....ls stuck in D state.
>
> ... IMO, far too much **tinkering** of code is going on currently without hard data (other than 'it looks good'), and this is exacerbating the problems.
>
> There are only two small behavioral bug reports I have received at this point, one of them looks like a bug that was in our networking before I began **tinkering** ;-)

Note that in these examples 'tinkering' and 'tinker' is always described in past or future tense: 'I was tinkering, ...too much tinkering, ...who want to tinker, I began tinkering...'. Thus tinkering is only expressible on the list as something that happens outside it – before or after the expressive and denominational work that is being done on the list itself. The list itself is outside the bounds of tinkering. We should remember however, that while tinkering is about local contexts and 'doable', material work, as a practice it also involves engagement with thinking, reflecting, and involves the mediation of representations.

Redesigning

Tinkering seems to stand in stark contrast to typical definitions of designing and redesigning, understood as practices associated with more logical and progressive types of work. In *Engineering and the Mind's Eye*, Ferguson (1992) traces the development of modern engineering practice. He sees this development as a move from the 'direct design' of the artisan (read: bricoleur or tinkerer) to the 'designing by drawings' of the modern engineer. In other words, while both the artisan and the designer actively conceptualize and manipulate the material world and therefore do design work, only the latter engages with specifically formal tools for representing the ongoing process. Ferguson is quick to point out that these are "differences of format rather than differences of conception" (Ferguson 1992: 5). He goes on:

> Usually, the 'big', significant, governing decisions regarding an artisan's or an engineer's design have been made before the artisan picks up his tools or the engineer turns to his drawing board. These big decisions have to be made first so that there will be something to criticize and analyze. Thus, far from starting with elements and putting them together systematically to produce a finished design, both the artisan and the engineer start with visions of the complete machine, structure, or device. (Ferguson 1992: 5)

By characterizing the difference between artisan and engineering work as based in the tools used to conceptualize design, rather than the conceptual tools used to materialize it, Ferguson does not end up 'primitivizing' the artisan. Just as the engineer analyzes and criticizes, so does the artisan. The difference in their work is linked to material difference rather than mental ability. Ferguson sees the work of designers as being in the forms of the model, the blueprint, and the formal drawing. The formalization of design representations is in no small part due to the need for greater coordination of redesigning activity, given the larger social contexts in which it takes place.

Here we can see the important role the LKML plays in providing the context for designing and redesigning practices. Below are some examples where 'redesigning' is being used on the LMKL:

> The one major downside, right now, is that Henry and Richard et al, keep talking about **redesigning** the klips structure to fit in with the more recent kernels better (ala netfilter, maybe). They've announced some *design specs* and I suspect that they would rather see the newer version of klips in the kernel tree than the crufty version that we are hobbled with in FreeSWAN right now.
> This is a last ditch deal-with-evil safety net system that has a fairly good chance of saving the data without extensively **redesigning** the *whole system*. Never said it was perfect.
>
> Now we get to the reason for this post. Has anything changed for 2.4.x? With release eminent, we don't really want to go through the **redesign** and implementation if the *architecture* is different for 2.4.x.

In these examples redesigning is understood as a practice that involves terms such as 'design specs', 'whole systems', and 'architectures'. Further, redesign is characterized as an ongoing practice, facilitated by conversation on the list. As something that occurs on and in the LKML, the practice of redesign thus requires representing the kernel in various ways in the online forum.

Again, it is important to remember that both tinkering and redesign work involves the use of representations, and that in both cases direct engagement with the artifact to be worked is to a greater or lesser extent 'deferred' while representations of it take primary focus. As noted above, in the case of redesign these include design models, flowcharts, blueprints and similar formal forms; tinkering on the other hand makes use of less formal representations. This may seem confusing, given that the main representational form is the source code that is being used in both tinkering and redesigning practice.

Shared digital objects: code as artifact, code as representation

Representations are used to mediate activity in all reworking practices; what differentiates the practices of tinkering and redesign is not that representations are used, but the purposes to which they are put. In particular it is important to address how software as both human-readable source code, and as compiled machine code, relates to the problems of representation and thus the issue of access.

Above I describe developer practices, ranging from tinkering as 'direct', off-list work on the kernel to redesign which involves the deferral of this immediate engagement with the kernel in favor of re-presentation through the intermediary of the LKML. However, both tinkering and redesign involves deferral. All activity is, in a sense, mediated. Thus, tinkering, like designing, is mediated by a number of different aspects, which include, but are not limited to mental representations, norms, language, and many other standards and tools. Differentiating, then, between tinkering and designing practices requires examining more specifically the kinds of artifacts and representations used at various times by the developers, and focusing more intently on how these artifacts impact developers' ability to engage in varying degrees of 'directness' and 'deferral'. What I want to highlight here is the different forms of mediation made possible by representations. That a tinkerer uses a representation in one way while a redesigner uses it in another means that differing representational aspects are highlighted and differing cognitive processes are engaged, but more importantly for questions of access, that *additional forms of distributed sharing become possible.* To better illustrate this, let me turn to the nature of software as artifact more generally, and more specifically, to its role in the development of Linux.

Artifacts and representation: objects and tools

As noted above, theories of human cognition that seem applicable to issues of access often place peoples' relationships with material or 'mediating' artifacts as central. In Activity Theory, for example, human actions are modeled as a triangle (figure 1) where a subject's goals are made possible through the mediation of tools.[19] However, as noted above, Linux kernel patches can be understood as both the goal of work as well as a means through which work takes place. While creating source code is arguably one of the main goals of work on the LKML, source code also serves as a mediating artifact for other goals, including the teaching of new developers or the coordinating of larger subsystems of Linux – objectives that are key to the ongoing coordinating aspects of the Linux community. Here it is important to distinguish between the kinds of tasks software accomplishes and the multiple ways it participates in different human activities.

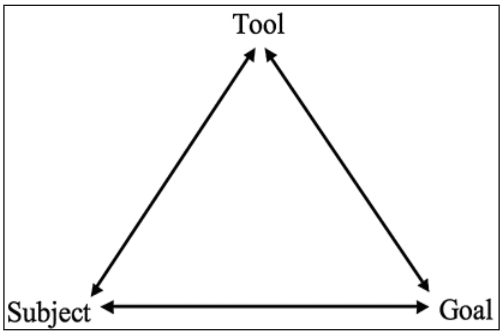


Figure 1: Activity Theory: mediating triangle.

One way to consider software as a tool is to see it as the means to particular ends. Figure 2 represents a typical software as tool relationship, such as is occuring now as I use word processing software to write this article. This occurs similarly in the case of the Linux Kernel when I use Linux as the OS for my computer (figure 3). In both these cases, the source code, the human-readable 'recipe' of software, has been compiled into a machine-readable form, made up of object files containing 'machine code' the instructions that are actually being executed on the computer hardware. Alternatively, we can understand software itself as the goal of work when developers work to maintain Linux, using applications such as text editors, and utilities like 'patch' (see above) to trouble-shoot and extend the Linux source code (figure 4). In this case, in order for Linux to work and operate as part of a computer operating system, the source code patch that was created must first be compiled into machine code before it can be executed on the computer and become a tool as originally depicted in Figure 4.

Based on this explanation, we might understand the distinction between software as goal and software as tool as predicated on whether or not it has been compiled. That is, software is a tool when it has been compiled into machine code, and software is the goal of work when it is in a source code form.[20] However, it is obvious that when source code is being used as an example, the source code itself is mediating the more community-oriented activities, rather than being merely the object of work. In the Linux community, as in most coding groups, source code snippets and patches are often used by developers to ask questions and to illustrate correct coding behavior (Ratto, 2005a: 9-11). In these cases, source code is transformed from something upon which work takes place into a tool used to represent, expand upon, communicate, and transform that work. It is safe to say that while machine code is primarily used as a tool, source code can be either a goal or a tool – distinguishing between these forms requires examining the practices to which it is being put.

What differentiates machine code and source code is merely the labor and resources required to convert one into the other – like the difference between ice and liquid water, the distinction between software as object or as tool depends upon the moment when it is being used, and the purposes to which it is being put, rather then being based on any formal distinctions. Activity theory describes this changing quality of artifacts as the 'object-tool shift' (Engeström, 1990) and notes that most artifacts have a similar duality (Miettinen, 1998). A productive aspect of the Activity Theory argument is that tools shift into goals—become objects of attention in their own right—when disturbances and problems impact the planned work.

Turning to the study of scientific practice, a similar quality of artifacts is examined by Knorr-Cetina (1999; 2001) in her analysis of the work of scientists. Complementing Rheinberger's (1997) exploration of 'epistemic things' Knorr-Cetina defines epistemic objects as, '…any scientific objects of investigation that are at the center of a research process and in the process of being materially defined' (Knorr-Cetina, 2001:181). These objects are productive because of their 'defining characteristic,' they have a '...changing, unfolding character...lack of 'object-ivity and completeness of being.' (Knorr-Cetina, 2001:181) While Knorr-Cetina follows Activity Theory in understanding artifacts—in this case scientific objects insubstantiated as material artifacts—as moving between a stable 'technical object' and mutable 'epistemic object' formation, she identifies this aspect to be a result of the absences within them that have yet to be discovered.[21] As both Activity The-

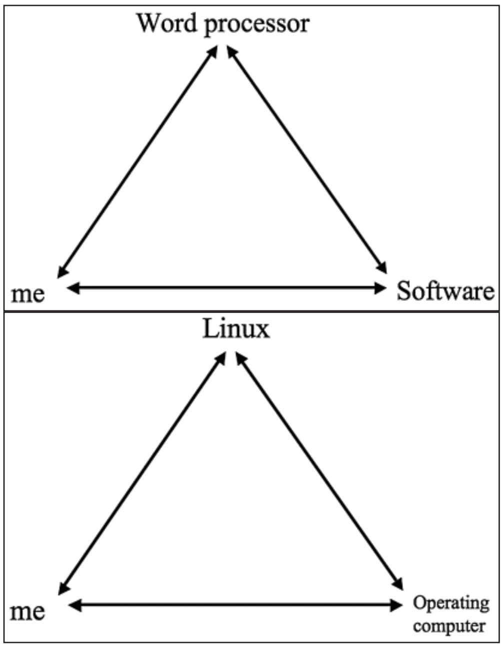
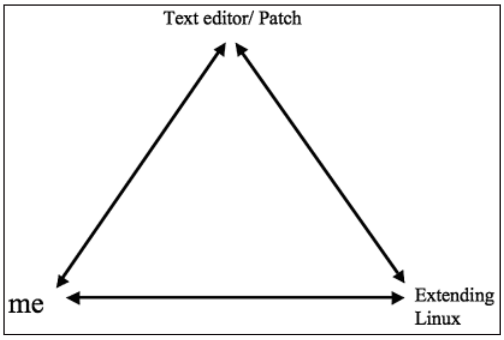


Figure 2-4: Software as tool: Linux as tool; Linux as goal.



ory and the concept of epistemic objects suggest, such qualities of material artifacts are only captured in the activities and practices of individuals and social groups. It is only through a material engagement with the artifacts in question that their relevant 'surplus' qualities are discovered and that their potentiality for knowledge acquisition and communication are revealed.

Leaving aside the more theoretical dimensions of these arguments, how can they help us better understand the work of the Linux developers and, ultimately, the work of distributed science and the idea of access? First, we should note a significant difference between Activity Theory's Object-Tool shift, and Rheinberger and Knorr-Cetina's descriptions of epistemic – technical objects. Whereas the latter focus most of their attention on how epistemic objects are stabilized and transformed into technical objects as part of the process of scientific work, Activity Theorists have tended to focus on the ways technical tools are destabilized and transformed into objects of work when problems occur. It is therefore vital not to confuse the binaries drawn by both theories; epistemic objects are not the same as the objects of work detailed in Activity Theory, nor are Activity Theory's tools the same as Knorr-Cetina and Rheinberger's definition of technical objects. Both theories highlight similar but ultimately different productive qualities; the theory of epistemic objects reveals the ways in which the surplus, unfolding qualities of the material world generate knowledge, while Activity Theory focuses to greater extent on the ways tools are generative of materially-productive work. This difference is one of focus rather than a problematic of the theories themselves – both theories provide insight into why and how objects are transformed as part of working practice. What connects both theories is an emphasis on the movement between two forms – how a stable tool becomes a destabilized object, or how an unstable epistemic object is transformed into a stable technical form. However, what is missing in the categorizations of both is a way of differentiating between types of artifacts and the specificity of the purposes to which they are put. In particular, both theories provide little purchase for thinking about a key issue for distributed communities, namely the ways in which action is coordinated over distance and over time. For this we must turn to an older theory of artifacts, one that provides a richer description of the variety of the representational qualities and purposes.

Wartofsky (1979) defines three classes of artifacts: primary artifacts are those used in material production and are typically thought of as physically existing; secondary artifacts are understood as representations of primary artifacts whose purpose is the transmission and preservation of existing modes of action and beliefs; while tertiary artifacts are considered 'imaginative artifacts', representations that encourage the reimagining of current activity. In a sense then, Wartofsky's theory extends the binary relationships described in Activity Theory and in scholarship on epistemic objects, by differentiating between their representational purposes. Wartofsky's theory provides a way of conceptualizing the three different aspects of Linux software; compiled Linux software as machine code can be categorized as a primary artifact used to do material work (i.e. to control computer hardware,) whereas Linux source code acts as both secondary and tertiary artifact, as secondary when it is used to represent current and past actions, and as tertiary when it is being used to extend and hypothesize about future activity. Each transformation involves particular kinds of individual and social resources and, equally, provides different modes of engagement for individual and social work.

To return then to Ferguson's definition of designing as engineer's work on blueprints and other physical artifacts which 'stand in' for and 'defer' the object being designed, we can characterize kernel development as requiring all three; first a primary artifact which is the focus and outcome of work (e.g. the compiled Linux kernel,) secondary artifacts that are used to reinforce and maintain existing kernel coding practice (source code snippets used as pedagogical tools,) but also tertiary artifacts which are used to conceptualize, represent and communicate new ways of coding and novel directions for kernel activity. Equally, we can now describe tinkering activity as mostly involving shifting between primary and secondary artifacts (the kernel and illustrative source code snippets that help with immediate programming needs,) whereas redesigning requires greater coordination between secondary and tertiary artifacts (source code snippets being used as illustrations for current activity and also as articulations of future possibilities.) An important reason for this difference is the greater need for coordination over longer periods of time entailed by redesigning practices.

'Tight' and 'Loose' – moving from individual to social conceptions of work

Above I noted how the Linux software shifts between roles, to be both objects of work and representations used to exchange coding knowledge and communicate and transform practice. An important aspect of the shift from primary to secondary to tertiary artifact, and the corresponding movement from tinkering to redesign practice, is that it mostly takes place on the LKML, within a community of other developers. Therefore, an initial response might be to characterize the movement to the list as a movement from an individualistic process of work to a group process. But tinkering, like all human activity, is never fully an individual act. Symbolic interactionism has pointed to the concept of the 'significant other' in order to capture this (Mead and Morris, 1974), Knorr-Cetina (2001) has expanded upon Rheinberger's (1997) definition of 'epistemic things' to address how they are embedded in 'epistemic cultures', and the notion of 'activity' in Activity Theory itself is predicated upon historically (and thus socially) generated communities, norms, and rules. While it should be noted that in this last tradition, the separation between 'practice' and 'activity' is based upon the extent to which the work being described is more or less social, more or less inherently within a community, maintaining this separation is often quite difficult. Becker (1982) points to the difficulty in seeing any human work as singular. In one example, Becker demonstrates the sociality of Van Gogh's painting (a seemingly wholly individualistic act) by articulating the work of others to provide the paint, the brushes, and the canvas. In one sense, then, each brush stroke, each daub of color can be described as a social activity. While it is ridiculous to describe Van Gogh's involvement in the painting of 'Starry Night' as equivalent to that of his brush-maker, it is important to acknowledge this more minor participation as legitimate and effectual. While it is possible that the participation of such-and-such brush-maker may not be indicative of a unique contribution (Van Gogh could possibly choose other brushes and other makers with impunity,) Van Gogh's access to tools must have an effect on the ultimate result of his work. Such a perspective is thus in line with the previous problematization of the notion of direct engagement. Just as work is never truly 'direct', it is also never only 'individual'.

How can we then characterize the shift we want to describe, from tinkering sorts of work to design practice? An alternative is to examine the organizational 'style' of tinkering and redesign, using the terms 'tight' and 'loose' rather than 'individual' and 'group' work. Such a shift in naming focuses on the different organizational and communicative needs of workers in these differing contexts, rather than positing this difference as based in the problematic and unreflective assumptions about human activity noted above. 'Tight' and 'loose' have long been used by engineers to characterize technical systems. More recently, Charles Perrow (1999) uses them to refer to differing organizational qualities of socio-technical infrastructures – aggregations of people, communities, and artifacts.

Using this nomenclature, tinkering is a more tightly organized and stable practice, characterized by a coherence of time and place. It is not that tinkering is organized from 'above' or that its rules are more explicit or formal – quite the contrary. Instead, the organization of the practice of tinkering is often 'self-organized' and results in a coherent set of relations between the people involved, driven, to be sure, by the simultaneous nature of the communications between them. In fact, tinkering work mostly takes place within and by groups of closely temporally connected workers. Although these relations may change from location to location and from context to context, for the moment of the tinkering practice they remain stable.

What the move to the list entails is thus a move away from the organizationally more 'tightly' connected tinkering work, and towards a more 'loosely' organized development effort. However, the term 'loose' should not be taken to mean that the rules, positions, or standards of the group are haphazardly decided or enforced. Instead, 'loose' indicates a spatial as well as a temporal distance between rules and enforcement, decision and adoption, tests and results. To use the metaphor of a rope, it is not that the knots are loosely tied, but that they are distantly spaced apart. Such spacing results in a sort of 'wiggle-room' that allows for different sets of problems—and different kinds of solutions—to arise within the groups involved. This is made more clear by the following use of the term on the LKML:

> The real question is why can't we just open 2.5 and only fix the VM to start with? Leave the kernel at 2.4.1pre10 and possibly use the -ac VM code (which has diverged from mainline), and leave people (Alan, Ben, Marcelo, et. al.) who want to **tinker** with it in small increments and do the drastic stuff in 2.5.

The key point here is that tinkering involves leaving people alone and allowing them to work with the shared objects in smaller increments. Tight contexts, then, are characterized by self-organization and stable relations among groups who are working in close temporal, if not geographical alignment.

Thus, while tinkering can be understood as a practice more tightly organized by space and time, redesigning, particularly in view of the whole collective of Linux development, can be seen as a more loosely organized process. The 'looseness' of the development process is often revealed when people on the list talk about 'redesigning'.

> ...That being so I'd like to run my current thoughts for **redesigning** the ppp support in the Linux kernel past people on this list.

> ...I suppose you could argue that **redesigning** Linux every few years is innovation, but unfortunately it's the same cast of characters doing it, so its not very innovative.

Redesigning can thus understood as a practice that requires certain kinds of consensus in order for its results to be accepted and used. Equally, this consensus can only be reached through communicative practices that involve large-scale decisions that must be agreed-upon by the relevant members of the LKML community. Unlike tinkering, where more informal and short-term agreements are sufficient, redesign requires agreements that last over longer periods and involve less coherent organizational contexts.

Access to distributed work as 'double-shift'

The activities of Linux development are various and complicated. However, two aspects stand out even in this brief analysis: first, that development activity requires the ability to shift between code as an object of work and code as representation, and second, that this shift is made necessary by the need to communicate and share work within the confines of the LKML. Therefore, access to Linux development, at least in so far as this includes the ability to contribute and make use of the shared resources of the development effort, requires two conjoined shifts; a shift from the 'tight' temporal organization of tinkering work to the 'loose' organization of redesigning, and a corresponding shift in the objects and representations involved. However, it is important here to note that not all Linux development activity requires all these shifts, nor do the practices of tinkering and redesigning outlined in this paper subsume all possible reworking practice. My goal here is not to describe the totality of development practice, rather this extended conversation is intended to create a working practice-model of access that will provide insight into the ways in which participation is mediated in collaboratory environments. In order to add additional detail and turn the more theoretical conversation into something more concrete, in the section below I use the above concepts to draw out a chart of reworking practice.

III. Experiment: charting the practices of Linux kernel developers

The diagram (figure 5) visually represents the concepts explored above in order to make them more applicable to problems of access. The X axis represents the three modes of artifact described by Wartofsky, while the Y axis describes the two extremes of organizational quality. I place the two reworking practices, tinkering and redesigning, on a third, Z, axis, with tinkering being linked most closely with primary artifacts within a tight contexts, and redesigning being focused on tertiary artifacts in a loose context.

What insights are possible by viewing Linux kernel development on this field? To return to an example previously mentioned, I will detail some of the steps involved in a successful patching project. While this is a relatively limited description of the complexity of shared development, these steps, and the kinds of practices and objects that are required, are illustrative of the more complex activities of Linux development.[22]

In the example briefly described in section II, the patch author (A) began by posting an introduction and his patch to the LKML:

> (A)This is the first public release of my libata compatible low level driver for the Marvell SATA family. Currently it successfully runs in PIO mode on a 6081 chip. EDMA support is in the works and should be done shortly. Review, testing (especially on other flavors of Marvell), comments welcome.
> (code continues…)[23]

Immediately, the subsystem organizer (B) states his willingness to put the patch 'upstream', meaning to move the related source code into the official kernel source tree. However, before this can happen, other developers (C & D) begin testing and extending the patch, exploring how it works and, in conversation with the original developer (A), attempting to solve problems:

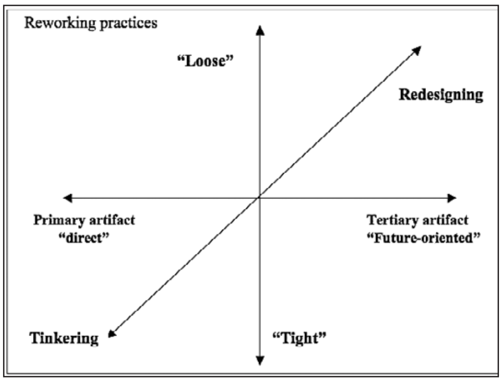

Reworking practices
"Loose"
Redesigning
Primary artifact "direct"
Tertiary artifact "Future-oriented"
Tinkering
"Tight"

Figure 5: Diagramming tinkering and reworking, the artifacts involved, and the organizational context.

> (B) Even though it's only PIO, if you feel this is stable, I would like to get it into upstream soonish. Current version looks OK to me.
> (C) (*Quotes source code, using it to demonstrate problem.)*
>
> (A) Some (non-functional) cleanup modifications since the version 0.10 driver I sent out 2005-08-30. (*code follows*).
>
> (D) First of all, thanks! I've been waiting for such a driver to appear…All tests are with the UP kernel. The hardware is an Asus PSCHSR-A board with Adaptec AIC8110 (*code snippets and explanation of attempted actions and problems follow*)…

In this process, sections of the patch code are referenced and rewritten, posts go back and forth between developers, and a new, slightly changed patch emerges. This shift is presented in the figure 7:

Soon after this, another developer (E) posts some suggested changes to the patch code, including the following statement:

> (E) please don't include 'scsi.h' in new drivers. It will go away soon.
> Use the <scsi/*.h> headers and get rid of usage of obsolete constructs
> in your driver.

This comment refers to the appropriate way to reference (e.g. 'include') other sections of kernel source code in the routines of the patch. It begins a larger conversation, with the subsystem maintainer (B) rejecting this comment and saying that the original coding syntax is correct:

> (B) It [the scsi.h include] stays until the rest of the libata drivers lose the include.
> After ATAPI support is done, I can stop 2.4.x support, and this and several other compatisms will go away.

Here, the subsystem maintainer (B) is referring to future needs and directions in the community, stating explicitly that the scsi include needs to stay until other coding work is accomplished. This conversation continues for a few more posts, with the subsystem maintainer (B) and the critical developer (E) debating issues of future direction of coding effort in the community and organizational responsibility, each using code from the original patch to illustrate their points. Ultimately a decision is reached regarding the patch and it is incorporated into the larger code tree.

The two shifts depicted in figures 6 and 7 are driven by the need to overcome a lack of relational coherence between organizational contexts and the concomitant transformation that must occur in the objects and representations of work. The first shift occurs because of the need to reconceptualize and communicate the action and artifact by representing it in a larger context. In this case, the mental or informal representations associated with tinkering practice in a tight context are not enough and must be supplemented with more deferred representations. This makes possible the incorporation of others into the problem, a shift to a more loosely organized context, and the movement from primary to secondary artifacts (1st shift). While the more pragmatic problem is solved at this stage, e.g. the patch code is extended and fixed, a new issue occurs, namely, the relationship between the current code and current coding behavior and how these relate to larger sections of source code and future needs. This engenders another shift, from the patch code as a secondary artifact used to represent current behavior and needs, into a tertiary form where it is used to describe and convince others of the need for other, future-oriented changes, in an even wider (looser) organizational context, (2nd shift).

Ultimately, the patch file is included in the official network subsystem file tree indicating a successful 'redesign' of both the patch code and the Linux kernel. This occurs because of the successful navigation of the full Z axis line of reworking practice, including both

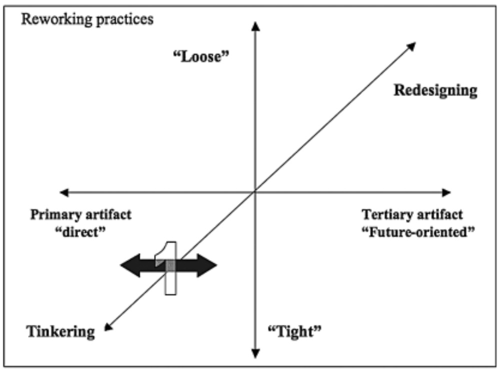

Reworking practices
"Loose"
Redesigning
Primary artifact "direct"
Tertiary artifact "Future-oriented"
1
Tinkering
"Tight"

Figure 6: Chart of first shift, from primary to secondary artifact, from tighter to looser organizational context.

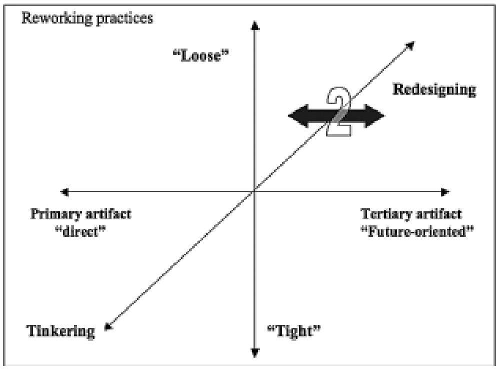

Reworking practices
"Loose"
2
Redesigning
Primary artifact "direct"
Tertiary artifact "Future-oriented"
Tinkering
"Tight"

Figure 7: Chart of second shift, from secondary to tertiary artifact, and from loose to looser organization context.

tinkering and redesigning modes of engagement. What makes this movement possible are two corresponding 'double-shifts', the movement between tight and loose contexts, and the transformation of the source code patch between primary and secondary, and secondary to tertiary forms.

The above analysis reveals that the ability to shift between the types of activities and forms of artifacts is essential to the ongoing nature of Linux development. Further, the analysis indicated that one reason for the necessity of shifting had to do with the coherence or discontinuity of the artifacts involved in the work. This latter aspect was indicative of the complex artifactual ground upon which the work took place. Key aspects of the expertise of the Linux developers involved their ability to both shift Linux from being a tool to being an object of work itself and to navigate between Linux source code as example of current activity, and as a vision of future needs. Equally, this shift also involves a shift from a tightly-coupled organizational context to the distributed and loosely-coupled space of the LKML. Focusing in on the two 'double-shifts' described above can thus provide a way of understanding successful or unsuccessful attempts at access.

IV. Relevance for Cyberinfrastructure and e-science

At the start of this article I made the claim that static models of access are less useful for examining and diagnosing issues with distributed scientific work than a more dynamic model that focuses on the working practices of the scholars and scientists and the artifacts with which they engage. I described current models as conceptualizing access as a series of preconditions that must be met in order for individuals or social groups to be able to 'reach' that which is to be accessed. Such preconditions include managerial, economic, technical, and social factors, each of which must be addressed and considered when access problems emerge. Seen in this light, access and more importantly, access problems, are understood as evoking a series of hurdles or gates, which must be overcome in order to make resources available.

Using the Linux kernel development effort as a exemplar of distributed, online work, I traced what an alternative, more dynamic model of access might reveal, simultaneously redefining the contexts and artifacts of such work in order to call attention to the specificity of activities and their related objects and tools. The resultant model conceptualizes access in distributed contexts as requiring the resources to manage two 'double-shifts'. The first double-shift involves moving from tight to more loosely organized contexts of activity and a corresponding shift in the object of work from a primary to a secondary form, e.g. from an artifact used to do work to a representation used to convey current activity. The second double-shift involves a further move to even more loosely organized contexts (particularly in regards to time), and a corresponding shift from secondary to tertiary form, e.g. from an artifact used to represent current activity, to one used to describe and visualize future needs.

These shifts, then, provide a novel way of understanding the issue of access, particularly in regards to contexts that involve the 'socio-spatial expansion' of the object of work (Engeström, 2001). The joint development of the Linux kernel, accomplished via the distributed form of the LKML, is a good example of such a phenomenon. However, as described in the introduction to this paper, more and more scientific and scholarly work is being done in a similar, distributed manner.[24] This has also been remarked on by Knorr-Cetina in her explorations of 'epistemic objects' noted above. Access as practice, specifically in the context of expert, distributed work, can thus be analyzed as the ability to link complex representations in both concrete and abstract spaces to the objects that they refer to, and, additionally, to leverage the use of these representations in various organizational contexts. An analysis of the material tools and communicative rules can provide ways of examining and diagnosing problems of access. Moreover, examining how individuals shift from 'tight' to 'loose' organizational contexts (rather than from individual to group work) and from conceptual to material models (rather than from concrete to abstract thinking) can provide insight for scholars working to better understand how distributed, shared, and collaborative resources are managed. For cyberinfrastructure, e-science, and e-scholarship in particular, two main insights are important.

First, the practice-based model of access described above makes clear that supporting distributed scientific or scholarly work requires supporting activities in both local and distributed contexts, e.g. both online and offline, but also requires assisting scientists and scholars in transitioning between these two modes of work. Pragmatically this means that in addition to developing online archives or databases, and offline applications and analytic tools, e-science developers should think explicitly about how users will track and represent the work they do in local spaces as well as in shared online spaces. Being able to communicate to others in the distributed context what has been done locally and in the past is a key element of shifting between 'tight' and 'loose' work contexts.

Second, and more importantly, this model of access demonstrates that attention to the objects of work themselves and the ways in which they are represented in online spaces is of paramount importance. Such an insight is empirically made in Beaulieu's analysis of digital databases on brain imagery (Beaulieu, 2004) and is also explored in work on bio-informatics and analyses of molecular biology and genetics, (e.g. Thacker, 2004; 2005; Ratto, 2006; Ratto and Beaulieu, 2007). These analyses make clear that in biological work in which digital information plays a key role, the relationship between digital representations as both tools for work and objects upon which scientists work, and the supposed real-world referents that make such work meaningful, are far from obvious.

However, the argument of this paper extends previous work by exploring how artifacts are used to represent future needs and behaviors. What the rework of the Linux kernel developers makes clear is that it is not just a relationship between the Linux kernel as compiled 'tool' (the 'real-world' referent in this case) and the kernel source code (representational 'object of work') that is at stake, but that relationships between various representational forms must also be negotiated. In other words, it is not just that the kernel as operational software is linked to its online depiction as source code, but that the relationship between online and offline representations of the kernel as source code must also be negotiated, in particular when the code is used to depict or represent future needs and decisions. One of the most important tasks for people involved in joint, distributed work is to negotiate not just how inscriptions or scientific representations act as mimetic depictions of some real-world referent, but also how representations are related to each other. That the multiplicity of modes of engagement between scientific representations is of particular interest is demonstrated by recent work in the philosophy of science on representation, instrumentation, and modeling that describes its more practice-based and active nature (e.g. Morgan and Morrison, 1999; Radder, 2003; Knuuttila, 2005) and notes in particular, the complexity of mediational activity involved.

Such an insight belies the more general trend in e-science and cyberinfrastructural development to sublimate the different scholarly and scientific objects under the more general rubric of 'data' and to focus on the similarities rather than the differences in scientific and scholarly practice. (Atkins, 2003). What the above model tries to make clear is that attention to the variety of scientific and scholarly practices and representational modes involved in doing distributed science is required. Such an insight follows from the more ethnographic oriented analyses of e-science (e.g. Hine, 2006), case study-based research (e.g. Edwards *et al.*, 2007), as well as theoretical explorations (e.g. Wouters and Beaulieu, 2006; Schroeder and Fry, 2007). In other words, while it is true that when scientific and scholarly objects are transformed into data they do tend to look the same, understanding how the data objects are transformed into various types of artifacts as part of the negotiation of tight and loose organizational contexts, over space and across time, is of paramount important when designing, maintaining, and using distributed systems. Access to distributed contexts and the issues of coordination this entails remains a problem for e-science (Cummings and Kiesler, 2005; Sonnenwald, 2006, as cited in Schroeder and Fry, 2007: 8). Overcoming these issues requires attention to how the artifacts involved in scientific work are part of the processes of coordination that make joint activity possible.

Conclusion

In this paper I used a longer-term distributed work process, that of Linux kernel development, as a case for exploring how access is productively negotiated, ultimately developing a chart of reworking that serves as a model for distributed access. This exploration involved four aspects; first, suggesting that FLOSS development provides an interesting view into distributed scientific and technological work in terms of joint, materially mediated practices of re-working. Second, defining a spectrum of re-working practices and noting the tools and objects of work that are required and created by these practices. Third, tracing out how digital objects served as both tools, objects, and as visions of a desired future. Fourth, describing the artifactual and contextual 'double-shifts' that are required to participate in shared, distributed work. Since science involves, as Latour (1987) has famously stated, the movement of inscriptions, a focus on the materialized representations of scientific work seems necessary, particular in cases of mutable, digital representations.

Studies of traditional scientific practice, such as those of Rheinberger and Knorr-Cetina described earlier, have examined the context of the laboratory and concentrated on the ways scientific representations are made to stand in for the objects they purport to represent, including, but not limited to real-world phenomenon and the results of scientific apparatus. Science in a distributed context is equally based on representation and inscription but seems to put even more pressure on the ways representations as epistemic objects in and of themselves, carry with them the forms and modes of engagement seen as appropriate for the communities involved in their construction and extension. If Linux is any kind of example, the explosion of artifactual shifting and transformation that digitality encourages becomes a focal point for addressing issues of access to resources in these contexts. However, artifacts do not (normally) transform themselves, but are manipulated by individuals within these communities according to need. Examining how people within distributed communities rely on artifacts to shift from tightly to loosely coordinated activity, and from immediate to future-oriented work, requires better understandings of how artifacts themselves act as centralizing resources for accomplishing material goals, help organize training, education, and normalization of community behavior, and also act as visions for future directions and possibilities. What makes science in distributed cyberinfrastructures 'accessible' is the ability to engage with the objects and artifacts of this work in all their various guises.

Acknowledgements

I would like to thank Tarleton Gillespie, Sampa Hyysalo, Tarja Knuuttila, and the anonymous reviewers for their detailed notes and advice. In addition, Chandra Mukerji, Yrjö Engeström, Anne Beaulieu, and Paul Wouters gave comments on early drafts.

Notes

1 I realize that the strategic importance of naming can not be underrated, particularly given the somewhat contentious nature of arguments about credit in regards to the Linux operating system and the role of previous coding efforts in its creation. (e.g. GNU.) However, to emphasize clarity I use the term 'Linux' to refer to only the kernel software, one part of an overall operating system. To refer to the full operating system based on the Linux kernel I use the phrase 'Linux operating system.'

2 LKML archive online at http://www.ussg.iu.edu/hypermail/linux/kernel/

3 From WordNet lexical database. Online at htto://wordnet.princeton.edu/

4 Estimate at The Linux Counter (http://counter.li.org/) February 8, 2005

5 The GPL enforces two main values of free/open source software (F/OSS) development. First, every program distributed under the GPL must include the underlying source code that makes it work. Second, and related to the first condition, the GPL requires that users be allowed to use the available source code to extend the original program or to create their own projects. A caveat to this agreement is that developers who make use of GPL source code must, in turn, also release the results under the GPL.

6 The term 'kernel' refers to the central component of an operating system, typically tasked with managing the relationship between hardware and software resources, with coordinating ongoing processes, and with managing access to memory. For more detail, see the wikipedia entry at http://en.wikipedia.org/wiki/Kernel_%28computer_science%29

7 This number, and the start-up date of 1991, both refer to the development of the Linux kernel, the 'heart' of the overall Linux OS. Calculating the complete size of the Linux OS is impossible, given the number of various parts of the OS

that are customized for different types of computer hardware, different purposes, and different users.

8 Although an article in the August 2003 issue of ComputerWorld magazine credited Linux with achieving a comparative 'user-friendliness' with Windows XP (Blau, 2003).

9 Posted to info-mini@udel.edu; from: Linus Benedict Torvalds; subject: Free Minix-like kernel sources for 386-AT.

10 Subject: The direction Linux is taking; Date: 2001-29 23:33:29 (LKML).

11 For more information about the patch utility, see http://www.fsf.org/software/patch/patch.html.

12 One of Eric Raymond's rules for open source software development. He calls it 'Linus' law' based on his understanding of Linux development practice. (Raymond, 2001).

13 Linux kernel version 2.6.20, released on 4 Feb 2007, is estimated to contain over 3.5million source lines of code or SLOCs. (http://widefox.pbwiki.com/Kernel%20Comparison%20Linux%20vs%20Windows, accessed 21/03/2007).

14 From 'Best practices for working with open-source developers', Ch.19 in (Raymond, 2003).

15 Subject: [PATCH 2.6.13] Marvell SATA support (PIO mode); Date: Aug. 30, 2005. (LKML).

16 Subject: Telecom Clock driver for MPCBL0010 ATCA compute blade; Date: Aug 30, 2005. (LKML).

17 For more on this notion and particular papers about it, see online at http://www.isrl.uiuc.edu/~gasser/dcp/ and http://www.limsi.fr/WkG/PCD2000/indexeng.html.

18 In addition to emphasizing the interactive aspects of technology design, scholars have noted that this work does not end in the laboratory or design studio. Users, in the moment of application, also productively work to 're-invent' technologies in order to fit them to the context as well as the task. (e.g. Rice & Rogers, 1980; Rogers, 1995). Von Hippel has described 'lead users' as early adopters who actively define possible uses as well as redefine and customize new technologies for novel purposes. (Von Hippel, 1984; 1994; 2001). Victor and Boynton have used the term 'co-configuration' to point to business processes predicated on close relationships between producers and users of commodities where the lines between the two are partially blurred. (Victor & Boynton, 1998). Finally, closer to home, Silverstone and Hirsch's edited volume uses the term 'domestication' to describe how the meanings and purposes of technologies are constructed (particularly in the context of the home) with the participation of users. (Silverstone & Hirsch, 1994; Berger et al., 2006).

19 For an overview of Activity Theory, its origins and recent developments, see The Center for Activity Theory and Developmental Work Research at the University of Helsinki's introduction, online at http://www.edu.helsinki.fi/activity/pages/chatanddwr/chat .

20 Such distinctions are made more complex by what happens when running software through debuggers and instruction set simulators, just-in-time compiling and software script execution – all processes that do not involve clearly separate coding and compiling steps, that blur the relationship between source code and machine code.

21 This point has also been made by Rheinberger, most recently in response to a critique by David Bloor (Bloor, 2005). In his response, Rheinberger wishes to make clear that the potentiality of technical objects to shift into epistemic ones is based not on their referential qualities, e.g. how their various characteristics might be named and described, but based on their surplus, their 'material transcendence' (Rheinberger, 2005: 406), how they exceed naming. Epistemic objects resist being turned into stable technical objects, what Activity Theory calls 'tools', "…by virtue of their preliminarity, of what we do not yet know about them, not by virtue of what we already know about them."

22 For a more nuanced examination of the fixing of bugs in an FLOSS community, see Sandusky & Gasser, 2005. For a brief outline of a possible method for automating the complex analysis required to understand how bugs are described, tracked, and resolved in FLOSS work, see Ripoche & Gasser, 2003.

23 Subject: [PATCH 2.6.13] Marvell SATA support (PIO mode); Date: Aug. 30, 2005. (LKML).

24 For specific case study examples, see Wouters & Schröder, 2003; Hine, 2006; more details on this general trend, see Wouters and Beaulieu, 2006; Thoutenhoofd and Ratto, 2007.

References

Arzberger, P., Schroeder, P., Beaulieu, A., Bowker, G., Casey, K., Laaksonen, L., et al.
2004 "Promoting access to public research data for scientific, economic, and social development." Data Science 3: 135-152.

Atkins, D. E.
2003 National Science Foundation Blue Ribbon Advisory Panel on Cyberinfrastructure (2003), Revolutionising science and engineering through cyberinfrastructure: Report of the National Science Foundation Blue Ribbon Advisory Panel on Cyberinfrastructure. http://www.nsf.gov/cise/sci/reports/atkins.pdf

Beaulieu, A.
2004 "From brainbank to database: the informational turn in the study of the brain." Studies in History and Philosophy of Biological and Biomedical Sciences. 35, 2: 367-390.

Becker, H. S.
1982 Art worlds. Berkeley: University of California Press.

Benkler, Y.
2002 "Coase's Penguin, or, Linux and the nature of the firm." YALE L. J., 4.03: 112.

Berger, T., Hartmann, M., Punie, Y., & Ward, K. J.(Eds.)
2006 Domestication of Media and Technology. Maidenhead, UK: Open University Press.

Blau, J.
2003 "Study: Linux nears Windows XP usability." ComputerWorld, August 04, 2003.

Bloor, D.
2005 "Toward a sociology of epistemic things." Perspectives on Science 13, 3: 285-312.

Bowker, G.
2000 "Biodiversity datadiversity." Social Studies of Science, 30, 5: 643-684.

Brooks, F.P.
1975 The Mythical Man-Month: Essays on Software Engineering. Reading, MA: Addison-Wesley.

Casey, K.
2003 "Issues of electronic data access in biodiversity." Pp. 41-64 in Wouters & Schröder (Eds.), Promise and Practice in Data Sharing. Amsterdam: NIWI-KNAW.

Cole, M.
1985 "The zone of proximal development: where culture and cognition create each other." Pp. 146-161in Wertsch (Ed.), Culture, Communication and Cognition. Cambridge: Cambridge University Press.
1996 Cultural Psychology: A Once and Future Discipline. Cambridge, Mass.: Belknap Press of Harvard University Press.

Coleman, E. G.
2005 The Social Construction of Freedom in Free and Open Source Software: Hackers, Ethics and the Liberal Tradition. Doctoral dissertation, Department of Anthropology, University of Chicago.

Collins, H.
1987 "Expert systems and the science of knowledge." Pp. 329-348 in Bijker, Hughes & Pinch (Eds.), New Directions in the Social Study of Technology. Cambridge, Mass.: MIT Press.

Cummings, J. & Kiesler, S.
2005 "Collaborative research across disciplinary and institutional boundaries." Social Studies of Science 35, 5:703-722.

Edwards, P., Jackson, S., Bowker, C. & Knobel, C.
2007 Understanding Infrastructure: Dynamics, Tensions, and Design. Report of a Workshop on History and Theory of Infrastructure: Lessons for New Scientific Cyberinfrastructures. January 2007.

Eklund, S., Feldman, M., Trombley, M., & Sinha, R.
2002 Improving the Usability of Open Source Software: Usability Testing of StarOffice Calc. Paper presented at the Conference on Human Factors in Computer Systems (CHI 2002), Minneapolis, MN.

Engeström, Y.
1987 Learning by Expanding. Helsinki: Orienta-Konsultit.
1990 "When is a tool? Multiple meanings of artifacts in human activity." Pp. 171-195 in Engeström (Ed.), Learning, Working and Imagining: Twelve Studies in Activity Theory. Helsinki: Orienta-Konsultit Oy.
2001 "Expansive learning at work: toward an activity theoretical reconceptualization." Journal of Education and Work, 14, 1: 133 - 156.

Ferguson, E. S.
1992 Engineering and the Mind's Eye. Cambridge, Mass.: MIT Press.

Fry, J.
2003 "The cultural shaping of scholarly communication on the Web: a case study of corpus-based linguistics." Paper presented at the Digital Resources in the Humanities 2003, Cheltenham, England.

Fujimura, J.
1987 "The construction of doable problems in cancer research." Social Studies of Science 17: 257-93.

Gasser, L., Scacchi, W., Penne, B. & Ripoche, G.
2003 Understanding Continuous Design in F/OSS. Proceedings of the 16th International Conference on Software & Systems Engineering and their Applications (ICSSEA-03).

Ghosh, R.A.
1998 "Cooking pot markets: An economic model for the trade in free goods and services on the Internet." First Monday, 3, 3. http://www.firstmonday.org/issues/issue3_3/ghosh/index.html.

Glaser, B. G. & Strauss, A. L.
1967 The Discovery of Grounded Theory: Strategies for Qualitative Research. Chicago: Aldine.

Glaser, B. G.
1992 Basics of Grounded Theory Analysis: Emergence vs. Forcing. Mill Valley, Ca.: Sociology Press.

Goldenberg-Hart, D.
2004 "Libraries and changing research practices: a report of the ARL/CNI Forum on e-research and cyberinfrastructure." Association of Research Libraries (ARL), 237:1-5.

Hakken, D.
1999 Cyborgs@Cyberspace? An Ethnographer Looks at the Future. London: Routledge.

Harper, D. A.
1992 Working Knowledge: Skill and Community in a Small Shop. Berkeley: University of California Press.
Himanen, P.
2001 The Hacker Ethic and the Spirit of the Information Age. New York: Random House.
Hine, C. (Ed.)
2006 New Infrastructures for Knowledge Production: Understanding E-Science. Idea Group.
Houghton, J. W., Steele, C., & Henty, M.
2003 Changing Research Practices in the Digital Information and Communication Environment: Department of Education, Science and Training (Australia).
Hutchins, E.
1995 Cognition in the Wild. Cambridge, Mass.:MIT Press.
Kelty, C. M.
2001 "Free Software/Free Science." First Monday, 6(12), http://firstmonday.org/issues/issue6_12/kelty/index.html.
2005 "Geeks, recursive publics, and social imaginaries.' Cultural Anthropology 20, 2: 185-214.
Kollock, P., & Smith, M. A.
1999 Communities in Cyberspace. London, New York: Routledge.
Knorr-Cetina, K.
1979 "Tinkering toward success: prelude to a theory of scientific practice." Theory and Society 8: 347-376.
1999 Epistemic Cultures: How the Sciences Make Knowledge. Cambridge, Mass.: Harvard University Press.
2001 "Objectual practice." Pp. 175-188 in Knorr-Cetina, v. Savigny & Schatzki (Eds.), The Practice Turn in Contemporary Society. London: Routledge.
Knuuttila, T.
2005 Models as Epistemic Artefacts: Toward a Non-Representationalist Account of Scientific Representation. Tarja Knuuttila. Academic Dissertation, University of Helsinki.
Latour, B.
1987 Science in Action: How to Follow Scientists and Engineers through Society. Cambridge, Mass.: Harvard University Press.
Lave, J. & Chaiklin, S.
1993 Understanding Practice: Perspectives on Activity and Context. Cambridge, New York: Cambridge University Press.
Lave, J. & Wenger, E.
1991 Situated Learning: Legitimate Peripheral Participation. Cambridge, New York: Cambridge University Press.
Lin, Y.
2004 Hacking Practices and Software Development: A Social Worlds Analysis of ICT Innovation and the Role of Free/Libre Open Source Software, Dissertation, University of York, September 2004.
Mead, G. H., & Morris, C. W.
1974 Mind, Self, and Society: From the Standpoint of a Social Behaviorist. Chicago: University of Chicago Press.
Miettinen, R.
1998 "Object construction and networks in research work: the case of research on cellulose degrading enzymes." Social Studies of Science 28, 3: 423-463.
Morgan, M. & Morrison, M. (Eds.)
1999 Models as Mediators. Cambridge: Cambridge University Press.

Moon, J. Y. & Sproull, L.
2000 "Essence of Distributed Work: The Case of the Linux Kernel", First Monday 5(11), http://www.firstmonday.dk/issues/issue5_11/moon/index.html.
Nichols, D. M., & Twidale, M. B.
2003 "The Usability of Open Source Software." First Monday, 8(1), http://www.firstmonday.dk/issues/issue8_1/nichols/index.html.
Nickell, S.
2001 "Why GNOME Hackers Should Care about Usability." In G. U. Project (Ed.) http://developer.gnome.org/projects/gup/articles/why_care/.
NIH
2003 Final NIH Statement on Sharing Research Data (No. NOT-OD-03-032): National Institutes of Health (NIH).
Norris, K. S.
1993 Dolphin Days: The Life & Times of the Spinner Dolphin. New York: Avon Books.
Nutch, F.
1996 "Gadgets, gizmos, and instruments – science for the tinkering." Science Technology & Human Values 21, 2: 214-228.
Pavlicek, R.
2000 Embracing Insanity: Open Source Software Development. Indianapolis, IN: SAMS Publishing.
Penchansky, R., & Thomas, J. W.
1981 "The concept of access: definition and relationship to consumer satisfaction." Medical Care 19, 2: 127-140.
Perrow, C.
1999 Normal Accidents: Living with High-Risk Technologies. With a new afterword and a postscript on the Y2K problem. Princeton, N.J.: Princeton University Press.
Preece, J.
2000 Online Communities: Designing Usability, Supporting Sociability. Chichester, New York: John Wiley.
Radder, H.
2003 The Philosophy of Scientific Experimentation. Pittsburg, PS: University of Pittsburg Press.
Ratto, M.
2003 The Pressure of Openness: the Hybrid Work of Linux Free/Open Source Software Developers. Unpublished PhD dissertation, University of California, San Diego, USA.
2005a "Embedded technical expression: code and the leveraging of functionality." The Information Society 25, 3: 205-213.
2005b "Don't fear the penguins: negotiating the trans-local space of Linux development." Current Anthropology 46, 5: 827–834.
2006 "Foundations and profiles: splicing metaphors in genetic databases and biobanks." Public Understanding of Science 14, 5: 31-53.
Ratto, M. & Beaulieu, A.
2007 "Banking on the Human Genome Project." In special issue on 'Genes' and Society: Looking Back on the Future, S. Z. Reuter and K. Neves-Graça, (Eds.), Canadian Review of Sociology/Revue Canadienne de sociologie, 44, 2: 175-201.
Raymond, E. S.
2001 The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. Cambridge, MA: O'Reilly.
2003 The Art of Unix Programming. Boston: Addison-Wesley Professional.

Rheinberger, H.
1997 Towards a History of Epistemic Things: Synthesizing Proteins in the Test Tube. Stanford: Stanford University Press.
2005 "A reply to David Bloor: toward a sociology of epistemic things." Perspectives on Science 13, 3: 406-410.
Rice, R. E. & Rogers, E. M.
1980 "Re-invention in the innovation process." Knowledge: Creation, diffusion, utilization 1, 4: 499-514.
Ripoche, G. & Gasser, L.
2003 Scalable Automatic Extraction of Process Models for Understanding F/OSS Bug Repair, Proceedings of the 16th International Conference on Software Engineering & its Applications (ICSSEA-03), Paris, France, December, 2003.
Rogers, E. M.
1995 Diffusion of Innovation. New York: The Free Press.
Rossi-Landi, F.
1983 Language as Work & Trade: A Semiotic Homology for Linguistics & Economics. South Hadley, Mass.: Bergin & Garvey Publishers.
Sandusky, R. J. & Gasser, L.
2005 Negotiation and the coordination of information and activity in distributed software problem management. GROUP '05: ACM 2005, International Conference on Supporting Group Work. Sanibel Island,Florida, November 6 - 9, 2005.
Scacchi, W.
2004 "Free and Open Source Development Practices in the Game Community." IEEE Software 2, 11: 59-66.
Schenkelaars, F. & Ahmad, I.
2004 Transparency and Accountability in the Public Sector in the Arab Region (Concept Paper 4 No. RAB/01/006): United Nations Online Network in Public Administration and Finance.
Schroeder, R. & Fry, J.
2007 "Social science approaches to e-Science: framing an agenda." Journal of Computer-Mediated Communication, 12, 2. http://jcmc.indiana.edu/vol12/issue2/schroeder.html.
Silverstone, R., & Hirsch, E. (Eds.)
1994 Consuming Technologies: Media and Information Domestic Spaces. London: Routledge.
Sonnenwald, D.
2006 "Collaborative virtual environments for scientific collaboration: technical and organizational design frameworks." Pp. 63-96 in Schroeder and Axelsson (Eds.), Avatars at Work and Play: Collaboration and Interaction in Shared Virtual Environments. Dordrecht, Netherlands: Springer.
Strauss, A. & Corbin, J.
1990 Basics of Qualitative Research: Grounded Theory Procedures and Techniques. Newbury Park: Sage Press.
Strauss, A., & Corbin, J., (Eds.)
1997 Grounded Theory in Practice. Thousand Oaks, Ca.: Sage.
Thacker, E.
2004 Biomedia. University of Minnesota Press.
2005 The Global Genome: Biotechnology, Politics, and Culture. Cambridge, Mass.: MIT Press.

Thoutenhoofd, E. & Ratto, M.
2007 Cyberinfrastructure and the cochlear implant: technological objects, social ordering, and epistemic conflict. Conference proceedings, Inside Knowledge, Amsterdam School of Cultural Analysis, March, 2007.

Torvalds, L.
2001 "What makes hackers tick? a.k.a. Linus's Law." Pp. xiii-xvii in Himanen (Ed.), The Hacker Ethic. New York: Random House.

Victor, B., & Boynton, A. C.
1998 Invented Here: Maximizing Your Organization's Internal Growth and Profitability. Boston, MA: Harvard Business School Press.

Von Hippel, E.
1984 Novel Product Concepts from Lead Users: Segmenting Users by Experience (Report 84-109). Cambridge, Mass.: Marketing Science Institute.
1994 The Sources of Innovation. New York: Oxford University Press.
2001 "Innovation by user communities: learning from open-source software." MIT Sloan Management Review 82.

Wartofsky, M. F.
1979 Models. Representation and the Scientific Understanding. Dordrecht: D. Reidel.

Wertsch, J. (Ed.)
1981 The Concept of Activity in Soviet Psychology. Armonk, NY: M.E. Sharpe.

Wright, M., Marlino, M., & Sumner, T.
2002 "Meta-Design of a Community Digital Library." D-lib magazine 8, 5. http://www.dlib.org/dlib/may02/wright/05wright.html

Wouters, P. & Schröder, P. (Eds.)
2003 The Public Domain of Digital Research Data. Amsterdam: NIWI-KNAW.

Wouters, P. & Beaulieu, A.
2006 "Imagining e-science beyond computation." Pp. 48-70 in Hine (Ed.), New Infrastructures for Knowledge Production: Understanding E-Science. London: Information Science Publishing.

Matt Ratto
Current affiliation:
Research Fellow
HUMlab & History of Ideas
University of Umeå, Sweden

New Affiliation, Summer, 2008:
Assistant Professor
Faculty of Information Studies
University of Toronto

email: matt.ratto@gmail.com