Versioning Dissonance:

Emerging Trends in Collaborative Production and Distributed Research

Jonah Bossewitch
SOCI G8200: Economic Sociology
Prof. David Stark
December 24, 2009

The most intense conflicts, if overcome, leave behind a sense of security and calm that is not easily disturbed. It is just these intense conflicts and their conflagration which are needed to produce valuable and lasting results.

Carl G. Jung, *The Structure and Dynamics of the Psyche*[1]

Less than a month before the 2008 US presidential elections, a small group of activists organized to develop a nationwide, cell-phone enabled, election monitoring system. This loosely connected network of volunteers was weaned on the free and open source cultures of wikis, barcamps, shared code repositories, and issue trackers. They succeeded in creating a functional monitoring system which attracted mainstream press cycles and helped identify multiple incidents of voter suppression. It is easy to imagine a system with these specifications costing the government millions of dollars and taking years to develop, but the group's do-it-yourself ethic and experiences led them to ingeniously leverage the latent infrastructure of the gratis microblogging service Twitter to power their monitoring system, and the Twitter Vote Report service was launched.[2]

The Twitter Vote Report project marked an important moment in the evolution of distributed cognition, collaborative production, and collective action. The project trained users to send properly formated text messages, allowing them to report information such as malfunctioning voting machines, voter suppression, and even wait times by zip code. After the information was submitted to Twitter, a very lightweight application slurped the data out of Twitter, presented it to volunteer "sweepers" for filtering, and transformed it into geo-spatial data visualizations. The project highlighted the power of aggregating and filtering numerous snowflakes of data and synthesizing them into cohesive narratives. It also helped popularize an important new style of collaboration percolating in free and open source software communities alongside *Distributed Version Control Systems* such as BitKeeper, Git, Mercurial, and Bazaar. By tracing the technical and cultural influences that birthed this project we can better describe the trajectories of these emerging styles of collaboration, and also identify important trends in ad-hoc community formation and collaborative production.

What socio-technical influences prefigured the Twitter Vote Report project? Does the Twitter Vote Report, and its close relatives, RapidSMS,[3] Ushihidi,[4] and SwiftRiver,[5] represent early crossovers of emerging styles of software code production to generalized content production? Will the next generation of collaborative production tools make Wikipedia seem centralized and hierarchical? Is *more information* a viable remedy for information overload?

In this essay I trace the history of version control systems leading up to the recent surge in popularity of Distributed Version Control Systems. I discuss the organizational and communication challenges that version control systems are designed to facilitate, and examine the interplay between these tools and their cultures of use. Finally, I consider some of the parallels between patterns in software development and patterns in content production and

---

[1] Jung, C. G., *The Structure and Dynamics of the Psyche*, 2nd ed. (Princeton University Press, 1970).

[2] A case study of the Twitter Vote Report was published by American University's Center for Social Media: Nina Keim and Jessica Clark, "Public Media 2.0 Field Report: Building Social Media Infrastructure to Engage Publics: Twitter Vote Report and Inauguration Report '09", Center for Social Media, American University. Updated October 2009. Available at http://bit.ly/tvr_report

[3] RapidSMS is "a free and open-source framework for dynamic data collection, logistics coordination and communication, leveraging basic short message service (SMS) mobile phone technology." It's development is paritially sponsored by Columbia's Earth Institue and UNICEF. http://www.rapidsms.org/

[4] Ushahidi is a humanitarian non-profit organization that is "building a platform that crowdsources crisis information. Allowing anyone to submit crisis information through text messaging using a mobile phone, email or web form." http://www.ushahidi.com/

[5] SwiftRiver is the project spawned when Ushahidi joined forces with the Twitter Vote Report project, and now describe themselves as a generalized crowdsouring platform, with a particular interest "in crisis reporting and international media criticism." http://swiftapp.org/

suggest directions for future research.

Code as Communication

Software, especially free and open source software (FOSS), is a strange socio-technical object. Software's inherent malleability combined with the standing invitation to reflexively modify it, make it difficult to get a firm fix on this complex non-human actor. Software code symbolically represents both data and instructions, and in this sense all (running) software is performative. Many software programs can be accurately described as having agency, although this agency is not yet autonomous. Coding is a form of expression used to communicate ideas and actions between humans, as well as machines. Writing and publishing software code is a kind of communication that is often an act of creative or political expression, and has even been ruled a form speech protected by the First Amendment by Federal circuit courts.[6]

Software production affords new models of collaboration, as individuals and organizations can collaborate indirectly through the intermediary object of code. Software's digital nature allows for easy copying and sharing, especially in a networked world. Its modularity and configurability also allows collaborators to avoid the administrative overhead of aligning resources and timelines by simultaneously supporting divergent requirements. Diversity can flourish without being dominated by homogeneity since abstractions allow the lowest common denominators of features to be shared, while domain specific customizations can be layered on top of intersecting requirements. Determining the precise contours and placement of these abstractions is a difficult architectural challenge that can require a great deal of communication, cooperation, and coordination. However, once these abstractions are established they can greatly expand autonomy and flexibility across the project.

Larger software projects can be understood as ecologies, comprised of the technological platform, the communities, and the processes that bind them together. This ecological model of software projects incorporates the dynamic lifecycle of the project over time, and provides insight into how participants might behave spontaneously, under complex and unanticipated circumstances. Large projects typically involve a diverse range of participants (developers, designers, project managers, users, vendors, clients, institutions, corporations, etc) who are connected to each other through formal and informal protocols. These protocols mediate interactions through structures ranging from legal contracts and decision making procedures, to technically mediated mailing lists and collaborative cyberspaces. In FOSS projects the tools used to manage changes over time are often built upon the edifice of the free software movement using many of the technologies and practices developed in earlier projects. This infuses projects with the values of the cultures they are built upon.

Christopher Kelty introduces the phrase “recursive publics” to describe groups that are concerned and capable of accessing and modifying the infrastructure supporting their existence.

> *A recursive public is a public that is vitally concerned with the material and practical maintenance and modification of the technical, legal, practical, and conceptual means of its own existence as a public; it is a collective independent of other forms of constituted power and is capable of speaking to existing forms of power through the production of actually existing alternatives.*[7]

Kelty demonstrates the political significance of software production, arguing that creating software is a political action similar to familiar political forms of expression like free

[6] McCullagh , Declan “Crypto Regs Challenged Again”, Wired Magazine, April 4, 2000. http://www.wired.com/politics/law/news/2000/04/35425

[7] Kelty, Christopher M., *Two Bits: The Cultural Significance of Free Software* (Duke University Press, 2008), p. 3.

speech, assembly, petition, and a free press. Since writing software is an act of creative expression, it is unsurprising that the artifacts created by a software community capture the values of that community through the inclusion (and omission) of features and the metaphors used in the software they create. Programmers habitually engage in the recursive analysis of meta-structures, and it is also unsurprising to see this analytical gaze turned back on itself. The community's proximity to the architecture of their own communication channels encourages a reflexive attitude towards their own communicative superstructure. This essay is not primarily focused on the overtly political dimensions of software communication, but instead focuses on cycles of innovation within communities that actively reflect on their own processes, workflows, and communication patterns.

Organizing Dissonance

FOSS ecologies are typically heterarchical, embodying a diverse range of individuals, organizations, and values, each bringing their own perspectives and "orders of worth" to the project.[8] Many successful projects attract participants that cut across a range of vertical sectors including freelancers, corporations, nonprofits, governments, and universities. Such a broad range of constituents inevitably bring together divergent priorities and requirements that can be difficult to reconcile. Successful projects must develop procedures for collectively negotiating the features and bugs to be taken into account, and how these activities should be prioritized and put in order. The ongoing maintenance of heterarchical organizations is a formidable challenge, exacerbated by software's ruthless finality–ultimately, an application either compiles or not; a binary reality applications share with the bits that constitute them. Software can be configured at runtime, or can fail gracefully, but at some point, if consensus is not reached, a disagreement becomes a "fork", and the project splits into two. Software is flexible, but failures are usually unmistakable, and broken software is worthless to almost everyone.

FOSS ecologies are sociological petri dishes and are a fertile breeding ground for experimentation on diverse models of structure and governance. The communities of practice that have formed around many software projects are constantly learning and adapting. Core areas of communal inquiry include improving access to knowledge within the community, refining procedures for conflict avoidance and resolution, facilitating consensus building, and fostering innovation – transparency, accountability, responsibility, and creativity. Discoveries and advancements in these areas of inquiry are regularly fed back into the ecology, and stabilized through processes and technologies which encode these improvements. These meta-goals are necessary for the continued success of the explicitly stated goals of the project, and they share a great deal in common with collaborative activities of all flavors.

Change is Our Maker

One of the most important tools used to orchestrate the complexity of software development is a *source/revision/version control system*. Since the early 1970s software developers have used these specialized systems to help manage and control changes to software code across contributors and time.[9] Even software projects with a single author benefit greatly from the disciplined use of a version control system. Experienced developers insist that using a version control system has transitioned from a "best practice" to a necessity.[10] Software teams

[8] For a good introduction to the concept of worth, "orders of worth", and heterarchy, see David Stark, *The Sense of Dissonance: Accounts of Worth in Economic Life* (Princeton University Press, 2009).

[9] Rochkind, M. J., "The Source Code Control System", IEEE Transactions on Software Engineering SE-1:4 (Dec. 1975), pages 364–370. http://basepath.com/aup/talks/SCCS-Slideshow.pdf

[10] The popular software pundit Joel on Software includes source control as the number one practice that well

that neglect this practice incur technical debts that often lead to mistakes and failure. The history of version control systems preserves a fascinating record of the collaborative production of complex information goods.

Managing multiple versions of code may not seem like a formidable challenge, but it is much more difficult than tracking multiple versions of natural language compositions. The structure of software code is much more rigid than human language since computers are bad at coping with ambiguity and compensating for errors. Engineering software is malleable when compared to engineering concrete or steel, but it is quite rigid in comparison to natural language and human-to-human communications. Software applications are composed of networks of inter-dependent elements, and even so-called "loosely-coupled" architectures are subject to syntactic and semantic constraints that are much more precise than natural language constructions. Changes in one area or section of a computer program might require significant adjustments to other areas of the program.

Software's malleability and adaptability translates into code that is in a perpetual state of flux. Code usually changes between release cycles, deployment sites, and runtime environments. Keeping track of this proliferation of differences is task that is easily susceptible to human error and ripe for automated assistance. Version control systems help software teams manage and maintain multiple variations of code, safely insulate experimentation from perturbing the existing system, and organize the division of labor between different contributors over space and time.

Maintenance: Often when a bug appears in production code, the development code base has advanced beyond that point. Bugs must be replicated in a development environment before they can be fixed with confidence, and it is crucial to recreate the precise state and conditions of the production environment in order to work on the problem. Fixes, or *patches*, must be carefully preserved and proliferated across the existing software installations, and propagated, or *ported*, backwards and forwards if it is determined that the patch should be applied to older and younger versions of the code.

Experimentation: Complex code-bases inhibit innovation if too much effort is required to introduce changes. Developers are often tempted to rewrite software than to modify or extend an existing legacy codebase, especially if the code is brittle. Similar to a rock climber's safety line, version control systems help ameliorate risk and anxiety by insulating a developer from the consequences of experimentation. The easier it is to set up a playground where ideas can be tried and tested, the more ideas will be generated. The easier it is to revert, or "roll back", changes to a known working starting point, the more conceptual risks can be taken. When developers learn to use and trust their version control system, the risks associated with making mistakes is greatly reduced, and radical modifications and refactorings can be attempted without fear.

Division of Labor: Large software projects often combine the knowledge of many different specialties, from information architecture and design, to systems administration, programming, and database development. Modern software applications are typically written in multiple programming languages, and require a working knowledge or a deep stack of technologies. It is common for projects to be divided between people with the requisite skills, and version control systems can assist with the division, delegation, and integration of these disparate contributions. The system serves the dual role of documenting and communicating changes between teammates. The communication channels are both direct and indirect, depending on how the system is configured and used.

Fearless of Commitment

---

managed team should adopt: http://www.joelonsoftware.com/articles/fog0000000043.html

In the early days, version control tools (e.g. SCCS and RCS[11]) were not network-enabled, and were only capable of tracking the history of changes to discrete files. They have progressed significantly, and are now used to help coordinate projects involving massive numbers of contributors, and ever more intricate experiments. Software continues to grow more complicated and vast, and teams are more distributed and dispersed. Accordingly, developers have reflexively refined their workflows and processes, and traces of these refinements are visible in the tools and workflows used to support these collaborations.

The most basic version control systems provide automated support for the storage, retrieval, logging, and identification of different versions of files. When an author decides to "commit" their changes to the system, they are prompted to log an explanatory message describing the changes. These annotations, as well as basic metadata such as timestamps and usernames, are stored alongside the substantive changes. Even these early systems commonly provided interfaces which allow the users to examine the exact differences between versions (and their metadata), and provided basic tools for merging different versions of files together. Early version control systems operated locally, on single files. They did not readily support the management of projects comprised of many files and directories, or provide support for coordinating teams of developers.

In order to better accommodate multi-file projects and teams of multiple collaborators, client-server version control systems were developed which stored centralized versions of the code, allowed for the bulk management of multiple directories and files, automated notification of transactions, and coordinated access to the canonical store. Two popular models of centralized source control management are file locking and concurrent access.

File locking systems allow many developers to simultaneously read and run the code, but only one person at a time can modify any particular file—files need to be "checked out" before they can be modified. Similar to borrowing a library book, only one person at a time can can work on any particular file, and no one else can work on that file until the borrower has returned, or "checked in", their completed changes. A major drawback of file locking systems is their technically enforced rigidity. If the person who checked out a file is unavailable, work on that file is effectively blocked, unless their lock is circumvented. Effective use of file locking requires a much greater degree of upfront coordination between team members to stay in sync with each other's working rhythms.

Concurrent systems allow more than one developer to modify files at the same time. While multiple people working on files simultaneously sounds like a recipe for dissonance and conflict, in practice concurrent systems manage changes with surprising fluidity. Most of the time, most development does not occur on the same file, and when it does, the changes rarely affect the same portion of that file. If the changes are simple, or complementary, the version control system attempts to merge the changes automatically. If the changes overlap, or become too complicated for the system to intelligently merge, the developer is alerted to the conflict and must manually merge the disparate changes to resolve the conflict. In the worst case, where a mistake does occur, it is possible to revert the code back to an earlier version reconstructed from the full version history.

Centralized, client-server, concurrent versioning systems scaled remarkably well through the numerous technological (micro)-revolutions of the 1990s and 2000s, and significantly facilitated the growth of peer production and the free software movement. The ubiquitous CVS (Concurrent Versions System) emerged in 1986, and started out as a suite of utilities wrapped around the single-file RCS system.[12] In 1999, at the height of the dot-com boom, VA Linux

[11] SCCS: Source Code Control System http://en.wikipedia.org/wiki/Source_Code_Control_System and RCS: Revision Control System http://en.wikipedia.org/wiki/Revision_Control_System

[12] http://ximbiot.com/cvs/wiki/CVS--Concurrent%20Versions%20System%20v1.12.12.1:%20Overview

launched SourceForge.net, a site offering freely hosted CVS, bug tracking, and mailing lists to any project whose code was released under an open license:

> *Ross Turk, SourceForge's community manager, remembers: "…When the site opened in November 1999, growth was respectable, if modest. At the time, the term "open source" was known only by those with a deep techie background. Though the site offered myriad free tools, only a small crowd of projects registered by the end of the year. That soon changed. By the end of 2000, SourceForge had thousands of projects registered; by the end of 2001, almost 30,000 were coding away. And the following year, the flood commenced. Since 2002, "we've been adding a hundred projects a day," Turk says.*
>
> *Fast forward to 2007 and SourceForge is now home to a sprawling universe of open source developers. It's an intense hive of software creators. Some 150,000 projects – and growing – reside there, covering every conceivable computing function.*
>
> *Just as important, SourceForge is the place to "see and be seen" if you're an up and coming open source project. It's developers chatting with developers, sharing, rubbing elbows, strutting their stuff, watching each other build.*[13]

Many individuals, teams, and projects had been running their own version control systems prior to the emergence of (what we would now call) cloud-based code repositories, but SourceForge marked an important turning point in the history of open source development and collaboration. The site set a new baseline for the minimal collaborative coding toolkit, and became an important watering hole which developers gathered around.

1999 also marked the release of the Subversion version control system (SVN)—an incremental improvement over CVS that smoothed out many of the rougher edges of CVS that made it cumbersome and increasingly unbearable for elaborate coordination. Conceptually, CVS and SVN were quite similar, but SVN offered improved support for operations that were so clunky in CVS that they were rarely used. Specifically, SVN made it much easier to reorganize the layout of a project, incorporate third party code, and also provided much better support for "branching" and merging. This support vastly improved developer's ability to avoid stepping on each other's toes, a growing problem as open source participation exploded along with the dotcom boom, GNU/Linux adoption, broadband penetration, and the relentless growth of World Wide Web.

SourceForge was slow to adopt the Subversion system, and many projects maintained a footprint of their activity on SourceForge, but hosted their actual repositories elsewhere. In the summer of 2006 , after a year of hosting its corporate open source projects on SourceForge, Google launched its own gratis project hosting service called Google Code.[14] The service was free of charge to any open source project, and initially offered subversion repositories and project management tools (bug tracking, wikis, and mailing lists) that were more user friendly than the tools on SourceForge.

The Discipline of Conflict

During this period FOSS communities were experimenting with various organizational structures around these tools. The Apache Software Foundation, an umbrella organization for many venerable open source projects, developed explicit bylaws outlining the meritocratic roles

[13] Maguire, James, "The SourceForge Story", Datamation, October 17, 2007. http://itmanagement.earthweb.com/cnews/article.php/12035_3705731_1/The-SourceForge-Story.htm
[14] http://googlecode.blogspot.com/2006/08/project-hosting-r-us.html

that contributors could earn.[15] Earning commit privileges on a project requires being nominated and voted into acceptance by existing commiters. The Debian GNU/Linux project developed an apprenticeship model of initiation, and has one of the most rigorous acceptance criteria in the FOSS world.[16] These strategies are very effective for maintaining shared values, quality control, and consistent licensing. However, they have also proved frustrating and sometimes stifling for faster moving, agile projects.

Some projects developed reputations for being snobby and elitist, requiring developers to jump through multiple hoops in order to earn their stripes. Extreme programming and dynamic languages continued to accelerate the pace of development, and it was common to hear complaints about stilted development practices, sluggish release cycles, and a regression to the creative mean.

Pockets of organizational innovation sprouted within these constraints. The Plone project is an open source content management system with a pluggable component architecture. The Plone community was generally more open and inviting than other enterprise class projects, but the commit rights to the core platform were still tightly regulated. Alongside the core platform, a project named the "Plone Collective" emerged as a centralized clearinghouse for plugins that could be installed on a Plone site.[17] The Plone Collective was run like a wiki for code—anyone could join and begin contributing to any product inside the Collective. A single SourceForge project was created, composed of multiple subprojects. This arrangement avoided the bureaucratic hassle of applying for a new project for every new product. It also overloaded the access control system and allowed all contributors to work on each other's projects. Projects within the Collective were rendered more visible to each other, and a great degree of trust was extended to all members of the Collective. Anonymous code contributions were not accepted, but anyone could apply for commit privileges to the Collective, and the barrier of entry was very low.

In a closely related development effort, the Zope3 community was also struggling to strike a healthy balance between inclusion and quality control. They developed guidelines around code submission that required that *all new development* must start out as a branch of the code.[18] Absolutely no new work could happen on the code repository's *trunk*. While branching was better supported in SVN than CVS, it was still a cumbersome operation that required more effort than simply applying a change to the code trunk. Typically, branching was reserved for radical experiments or backtracking to a much earlier version of the code, not for simple enhancements or bug fixes. The Zope3 project instituted this disciplined workflow to encourage experimentation without sacrificing the stability or integrity of the trunk. These arrangements are not enforced by the software, rather they are enforced socially, by peer pressure and convention.

Multiplicty and Social Coding

The Linux kernel, arguably one of the most important FOSS projects, was managed without version control system until 2002. Linus Torvalds the linux kernel project leader disliked centralized version control systems, which he considered unsuitable for kernel development. The Linux kernel is a very large and complex software project, has extraordinary quality demands, and also attracts thousands of developers. Changes were meticulously tracked through a distributed hierarchy of delegates, but the system was showing strain. In 2002 Linus finally decided that a "distributed version control systems" (DVCS) would match the project's

[15] http://www.apache.org/foundation/how-it-works.html#roles
[16] Coleman, Gabriella, "CODE IS SPEECH: Legal Tinkering, Expertise, and Protest among Free and Open Source Software Developers," *Cultural Anthropology* 24, no. 3 (2009): 420-454.
[17] http://dev.plone.org/collective
[18] http://www.zope.org/DevHome/CVS/ZopeDevelopmentProcess

needs. The Linux kernel was migrated to the proprietary BitKeeper versioning system, a selection which sparked great controversy because of its closed license.[19] In 2005, licensing disputes eventually led to the creation of freely licensed distributed version control system and the DVCS named *Git* was created.

Distributed Version Control Systems operate on a different model than repositories managed by a centralized, client-server, system. The DVCS model is peer-to-peer, and while it can be configured to resemble a traditional client-server transactions, it can also support more complex interactions. In a DVCS system, every developer works locally with a complete revision history, and changes can be pushed and pulled from any other peer repository. The version control system has vastly improved support for merging across multiple repositories, and all working *checkouts* are effectively *forks*, until they are merged back onto a canonical trunk.

The demands on the Linux kernel project prefigured the demands on other projects. In the past few years distributed versioning systems have dramatically increased in popularity.[20] *Mercurial, Bazaar,* and *Git* have emerged as the most popular open source DCVS systems, and hosting services have launched offering each of these systems free of charge for open source projects. Google Code began supporting Mercurial repositories alongside Subversion repositories in late 2009.[21] Cannonical, the company which sponsors the Ubuntu GNU/Linux distribution, offers free Bazaar hosting to open source projects on Launchpad.net. In February 2008[22] GitHub.com launched, a "social coding" site which provides Git hosting and rich social networking tools to all the developers using the site, gratis for open source code. Bitbucket.org offers similar social networking tools around *Mercurial,* and describes itself as "leading a new paradigm of working with version control".

The centralized hosts of peer-to-peer protocols broker a new balance between centralization and federation. They facilitate coordination, but do not mandate it. A site like GitHub can track and aggregate multiple branches of development, but branching does not require any permission or upfront coordination. Instead of requiring an upfront investment of attention and energy to coordinate development activities, DVCS concentrate on improving the mechanisms for developers to track, visualize, and merge changes. The costs of coordinating collaboration is deferred, and the communication overhead required to synchronize and align different branches of code is (hopefully) reduced.

There is a fascinating culture emerging around DVCS, facilitated by software, but responding to (and suggesting) shifts in collaboration styles. As one developer explains:

> *SourceForge is about projects. GitHub is about people... A world of programmers forking, hacking and experimenting. There is merging, but only if people agree to do so, by other channels... GitHub gives me my own place to play. It lets me share my code the way I share photos on Flickr, the same way I share bookmarks on del.icio.us. Here's something I found useful, for what it's worth... Moreover, I'm sharing my code, for what it's worth to me to share my code... I am sharing my code. I am not launching an open source project. I am not beginning a search for like minded developers to avoid duplication of efforts. I am not showing up at someone else's door hat in hand, asking for commit access. I am not looking to do battle with Brook's Law[23] at the outset of my*

[19] For a good collection of links to this controversy, see http://better-scm.berlios.de/bk/.
[20] Paul, Ryan, "DVCS adoption is soaring among open source projects", January 7, 2009, Ars Technica. http://arstechnica.com/open-source/news/2009/01/dvcs-adoption-is-soaring-among-open-source-projects.ars
[21] http://code.google.com/p/support/wiki/DVCSAnalysis and http://arstechnica.com/open-source/news/2009/04/google-code-adds-mercurial-version-control-system.ars
[22] http://github.com/blog/1-hotkeys-and-wikis
[23] Brooks's "law" is a software development principle states that "adding manpower to a late software project makes it later". Frederick P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, 2nd ed. (Addison-Wesley Professional, 1995).

brainstorm.[24]

Sometimes developers simply want to publish and share their work, not start a social movement. Sometimes they want to contribute to a project without going through masonic hazing rituals. DVCS facilitates these interactions far more easily than traditional centralized version control systems and the hierarchical organizations which tend to accompany these systems. Part of what makes this all work smoothly are very good tools to help merge disparate branches of work. This all sounds chaotic and unmanageable, but so did concurrent version control when it first became popular.

In anecdotal accounts of switching to DVCS, developer's describe an increase in the joy of sharing—the tools help reduce the focus on perfecting software for an imagined speculative use and the overhead of coordinating networks of trusted contributors. The practice really emphasizes the efficient laziness of agile programming, and helps people concentrate on the immediate requirements, not become preoccupied with endless planning and prognostications.

In some respects, this emerging style of collaboration is more free-loving than an anonymously editable wiki, since all versions of the code can simultaneously exist – almost in a state of superposition. Wikis technically support the preservation of diversity in a page's history, but in a centralized wiki the current page is the (ephemeral) final word. DVCS are developing richer interfaces to simultaneously represent diversity, and facilitate the cherry picking of features from across a range of contributors. The expression of a multiplicity of heteracrchical voices is explicitly encouraged, although there is a hidden accumulation of technical debt that accrues, the longer the merging of different branches of work is delayed. Of course, sometimes you may actually want to start a community or social movement around your software, which is still possible, but is now decoupled and needs to be managed with purposeful intent.

Torrential Flows, Swift Rivers, Great Lakes

It is easy to imagine the practices around DVCS percolating from code production to other areas of information production, similar to the ways that centralized, client-server, concurrent, version control systems influenced wiki culture. We are starting to see hints of this style of collaborative style breaking free from the software development world and into the realm of content production, initially in the forms of distributed research, filtering, and analysis. As Benkler has argued in *The Wealth of Networks*, ranking and filtering is itself just another information good, itself amenable to peer production, but the best ways to organize and coordinate this production, still need to be worked out.[25] Just as computing is becoming massively parallel, and in turn, spurring algorithmic innovation around MapReduce,[26] distributing *and reassembling* human labor is a problem facing an increasing number of networked projects. DVCS offers us an innovative and promising model for thinking about new styles of distributed collaboration.

Returning to the project introduced at the beginning of this essay, we are now prepared to recognize the significance of the Twitter Vote Report from a fresh perspective. The system represents an important moment in the popularization of the practice of networked distributed cognition. While flickr.com and de.licio.us enable distributed collecting and aggregating of photos and links, Twitter Vote Report demonstrates the possibility of aggregating arbitrary structured data. Participants glimmered the power of synthesizing a blizzard of data snowflakes into a meaningful bank of knowledge.

[24] http://kiloblog.com/post/sharing-code-for-what-its-worth/
[25] Benkler, Yochai, *The Wealth of Networks* (Yale University Press, 2007).
[26] MapReduce is the name of Google's programming model for processing larege data sets over distributed computing resources: http://labs.google.com/papers/mapreduce.html

In the months that followed the election, the Twitter Vote Report developers helped monitor the 2009 Indian general elections, and collaborated with the Ushahidi project, a free and open source project committed to building a platform for crowdsourcing crisis information. The project was rebranded *Swift* "a toolset [named SwiftRiver] for crowdsourced situational awareness":

> *Swift hopes to expand [Twitter Vote Report's] approach into a general purpose toolkit for crowdsouring the semantic structuring of data so that it can be reused in other applications and visualizations. The developers of Swift are particularly interested in crisis reporting (Ushahidi) and international media criticism (Meedan), but by providing a general purpose crowdsourcing tool we hope to create a tool reusable in many contexts. Swift engages self-interested teams of "citizen editors" who curate publicly available information about a crisis or any event or region as it happens.*

Twitter Vote Report learned during the election that in addition to gathering information, it was essential to filter it. The Ushahidi team had been working on a design for a curation workflow that would enable crowdsourcing the filter, alongside data collection. In their simple design, curators volunteer to rate incoming data from the social networking site *de jour*, publicly stating who they trust and the facts they believe.

Unlike the pure modernist visions of the semantic web, these systems of collaborative analysis attempt to represent the superposition of multiplicity of voices. Instead of a single overarching standard of value, determined in advance, the SwiftRiver platform presents a wide range of judgments, and concentrates on meaningfully aggregating this information, always preserving the originating source. While machine intelligences will play an active role in harvesting, inferring, and filtering, these agents operate alongside human intelligence. Data is always presented contextually, connected to its curator—there is no "what" outside the context of a "who."

The SwiftRiver platform is an early example of the style of collaboration emerging around Distributed Version Control Systems crossing over from code production to content production. This transition is characterized by a shift from centralized, hierarchical, canonical versions of knowledge giving way to federated, distributed, heterarchies. These knowledge communities strive to preserve the multiplicity of perspectives, and the system aims to aggregate, merge, and synthesize the diversity instead of collapsing it into a single, homogenized, view.

The Swift platform is focused on filtering hot flashes of news, but it is easy to imagine this platform being used to conduct collaborative analysis across many different kinds of data sets, archives, and corpora. The core idea of crowdsourcing the curation and analysis of information by annotating anything at the other end of a URL is applicable to many efforts beyond the filtering of news. SwiftRiver is precisely the kind of tool that could vastly improve the organization of investigations and analysis of every set of documents released on WikiLeaks[27]. More generally, Swift can also help transform qualitative data into evidence which supports an argument or hypothesis—an essential activity for social scientists, educators, journalists, activists, and lawyers. The data visualization mechanisms Swift implements could aid researchers across domains in the discovery of patterns and lacunae from the corpus they are commonly investigating, deepening their understanding of context and discourse. This approach moves beyond the machine-based aggregation of data streams, into synthesizing distributed human judgment.

[27] WikiLeaks is a whistelblower publishing platform run by a non-profit organization funded by human rights campaigners, investigative journalists, technologists and the general public. http://wikileaks.org/

Conclusion

There is a great deal to study and learn from the collaborative practices of FOSS communities. The history of version control systems teaches us to create conditions that support risk taking while minimizing consequences, to defer the upfront communications costs of coordination when possible, and to save everything. Finding a balancing point between centralization and federation is the holy grail of peer production. Units of work must be "just right", not too big, not too small, but still interesting and meaningful. To appreciate the socio-technical traces of collaboration and it is important to record these stories while they are fresh in our collective memories, perhaps before we recognize their full significance..

The adoption of Distributed Version Control Systems is growing quickly, but the systems are still quite new. Their rapid uptake suggests that they these systems satisfy important organizational needs and desires. If we consider the conceptual similarities of coding to other forms of information production and the recursive production capabilities of FOSS communities, it is likely we will soon see more analogs of Distributed Version Control Systems crossing over into all forms of information production. The Twitter Vote Report, and its successor, SwiftRiver represent early examples of this trend, but more will follow.