Collaboration with Lean Media: How Open-Source Software Succeeds

Yutaka Yamauchi[†] **Makoto Yokozawa**[‡] **Takeshi Shinohara**[‡] **Toru Ishida**
Department of Social Informatics
Graduate School of Informatics
Kyoto University
Kyoto 606-8501, JAPAN
yutaka.yamauchi@anderson.ucla.edu {m-yokozawa,t-shinohara}@nri.co.jp ishida@i.kyoto-u.ac.jp

ABSTRACT
Open-source software, usually created by volunteer programmers dispersed worldwide, now competes with that developed by software firms. This achievement is particularly impressive as open-source programmers rarely meet. They rely heavily on electronic media, which preclude the benefits of face-to-face contact that programmers enjoy within firms. In this paper, we describe findings that address this paradox based on observation, interviews and quantitative analyses of two open-source projects. The findings suggest that spontaneous work coordinated afterward is effective, rational organizational culture helps achieve agreement among members and communications media moderately support spontaneous work. These findings can imply a new model of dispersed collaboration.

Keywords
electronic media, distributed work, cooperative work, open-source, software engineering, CVS, innovation

INTRODUCTION

Geographically dispersed programmers create reliable and innovative software collaboratively over the Internet. Such software, called open-source software, now competes with that developed in software firms in terms of reliability and performance. From a traditional perspective, this would seem impossible as managing and leading software development have been one of more challenging and complex issues. Surprisingly, however, open-source software is usually developed by volunteers who usually do not demand monetary rewards. No formal quality control programs exist and no authoritative leaders monitor the development.

In particular, it is surprising that open-source development achieves smooth coordination, consistency in design and continuous innovation while relying heavily on electronic media. This 'electronic' environment imposes limitations due to the absence of face-to-face contacts. Project collaborators are limited to written communication and lack spontaneous discussion. They have little real-time knowledge about each other. Complex and ambiguous messages cannot be shared because of the limited interaction. Yet, open-source initiatives achieve outstanding results and demonstrate innovative product development. How is this possible? This study deciphers some of the paradox.

Practices of open-source software development are important to CSCW (Computer Supported Cooperative Work) research for two reasons. Firstly, software development is traditionally a coordination-intensive process and has attracted several CSCW researchers (e.g., [2, 8, 9, 15]). Secondly, open-source software is developed by geographically dispersed programmers with the aid of collaborative technologies such as electronic mail, the world-wide web and software configuration management systems. The use of these technologies has been the central focus of CSCW research.

This paper presents findings that show how open-source development avoids limitations of dispersed collaborations and address the sources of innovation in open-source. The study investigated two open-source projects with detailed observation of over 2,800 messages, interviews with several members and quantitative analysis of more than 1,000 messages.

OPEN-SOURCE SOFTWARE

Open-source software has been popularized only recently, though it has a history going back to the invention of computers. In the early days of computing software was distributed with the source code. This became less widespread when software venders began to conceal the source code. Commercialization of software provided user benefits but served to limit development or manipulation of software to meet special needs. Realizing this trend as a

[†] Current address: Anderson School at UCLA, 110 Westwood Plaza, Los Angeles, CA 90095

[‡] Visiting professors jointly working at Nomura Research Institute, Ltd. (http://www.nri.co.jp)

limiting factor in software development and even in progress of computer technology, Richard Stallman (a researcher of MIT AI Lab.) started a GNU (GNU's Not Unix) project to provide free UNIX. Over the years since then, GNU programmers have been creating a variety of innovative open-source tools.

The emergence of Linux, a free Unix-compatible operating system, changed many of the attitudes towards open-source software. Linus Torvalds, the creator of Linux kernel, succeeded in gathering a development team and contributors over the Internet. He frequently released new versions that were not necessarily stable and other programmers debugged them and added new features. Eric Raymond, an evangelist of open-source, labeled this approach the 'Bazaar' model connoting a chaotic but effective organization in contrast with the traditional 'Cathedral' approach [20]. He also explains that the stability of the open-source programs comes from the large number of developers and testers: Given enough eyeballs, all bugs are shallow. In addition, the frustration that the programs do not provide the capacity users want prompts them to add new features: Scratching a developer's personal itch.

The key feature of open-source software is the availability of the source code. It enables users with special requirements to tailor programs, skillful users to eliminate bugs they encounter in use, and innovative programmers to enhance programs even outside a formal project. The widely used free software license, GNU General Public License (GPL), guarantees that anyone can modify and redistribute programs insofar as he/she does not hinder others from modifying and redistributing them.

There are now an incalculable number of open-source software programs. Examples include Linux, FreeBSD, Apache, Perl, Sendmail, etc. Some are category killers, which corner the market leaving no room for commercial products. Vendors, such as Apple, Netscape Communications and Sun Microsystems are among the growing number of companies making use of open-source programs.

ELECTRONIC MEDIA AND COOPERATIVE WORK

Media in Open-Source Software Development

The primary communication tool used by open-source programmers is electronic mail. In particular, mailing lists or distribution lists are used to broadcast messages to all the members. HTML messages are not used since straightforward text messages are easier to manipulate. Files are usually sent encoded as text format and attached to messages.

In sending modification of source code, programmers do not send an entire set of source files. They use the tool 'diff' to extract the difference between an original set of files and a modified set. The modified version can be reproduced by a tool called 'patch' if the original version and the difference are at hand. These tools help to save on bandwidth since difference files are usually smaller than the full source-code files. In addition, since the difference files show modified lines of code with corresponding original lines, receivers can immediately know which parts were changed.

Another important tool used in open-source software development is Concurrent Versions System (CVS) [3]. CVS is one of the software configuration management (CM) systems, which store multiple versions of the source code and enable users to download and upload the code. Though designed for software evolution management, these are used as coordination systems to support articulation work in making members' involvement public [8]. CVS is different from other configuration management systems (e.g., Revision Control System) in that it enables the concurrent development. That is, multiple programmers can modify the same file at the same time. Yet conflicting modifications have to be merged later.

Web pages also play important roles in open-source projects. They enable potential developers and users to join the projects. Anyone can know the details of the programs and download and use them from web pages. In addition, web pages also introduce the mailing lists of the projects. Most of open-source projects have an interface to the archived messages with search features.

Characteristics of Electronic Media

While electronic media eliminate the spatial and temporal boundaries of communication, they bring limitations in other aspects compared with the face-to-face interactions.

Firstly, electronic media weaken the social presence of communicators: communicators cannot perceive whether partners understand their words and how friendly partners are [22]. Social presence, determined by technological capacities of the media, shapes the way people communicate.

Secondly, electronic media make it difficult to transmit equivocal messages, whose ambiguity in meaning permits multiple interpretations, because of the limited amount of communicative cues and sluggish interaction [6]. Face-to-face informal conversations are the richest medium and thus easily accommodate equivocal messages while written messages are more rigid and convey less information.

Thirdly, the media reduce social context cues, such as the place and time of the communications and the name, age, sex and title of the partners [24]. In case of electronic mail, people often send messages to those they do not know. Because of this, individuals tend to push their ideas in a stronger manner than they necessarily would in a face-to-face scenario.

Owing to the characteristics of electronic media, communications can sometimes suffer problems. It is difficult to achieve agreement among members [11, 12]. Human and social topics are relatively more difficult than computational and numerical topics [11]. In addition, communicators cannot build good relationships and tend to evaluate partners less favorably [13]. Uninhibited behaviors, such as swearing, insulting and hostile messages, happen frequently [12]. Finally, communications mediated via computers are inevitably formal because the initiation of informal communication requires physical proximity [14]. CSCW researchers have proposed several advanced tools challenging these limitations.

Assumptions about Software Development

We have several assumptions about software development to discuss the media use and coordination processes of open-source software development.

Resulting products can be so complex that developers cannot eradicate bugs. It is necessary to test programs frequently when new components are added and even small parts are altered. In addition, the insufficient discussion on the design at the outset inevitably brings serious flaws at the late stage of development, when it can be difficult to change. It is, however, impossible to fully describe product specifications based on the requirements analysis before starting implementation since requirements often fluctuate and conflict continuously [4].

Work in software development is a contingent process. There is no standard way of doing work and each development work is application specific. One cannot prescribe all the contingencies that will be faced before the work unfolds. Although this is the case for any kind of work in organizations, software development work faces an exceptionally large number of contingencies since the work deals with complex technologies.

Software development requires many people to be involved. If systems could be developed in a small team, or a community of practice, in which programmers were strongly tied, the development would not be so arduous. In communities of practice, members can learn something new efficiently, solve problems smoothly and find the opportunities of innovation [1, 28]. In contrast, the coordination across boundaries is inevitably formal and it is difficult to manage all the contingencies [15].

As a consequence, software development is a coordination-intensive process. By coordination we do not mean the decomposition of work and later aggregation. Rather, ad hoc and situated coordination is required in the face of uncertainty. Problems that emerge unpredictably in the course of actions have to be resolved through flexible coordination. The difficulty of software development comes from the frailty of the coordination. In case of large systems, coordination breakdowns seem inevitable. The huge amount of research on software engineering cannot provide complete solutions to this strenuous problem.

Research Questions

Based on the review of electronic media and assumptions on software development, we set three research questions.

1. How can open-source software development achieve smooth coordination using limited media?

From the review above, smooth coordination is difficult in electronically mediated environments. Software development, however, requires it and many examples of open-source development demonstrate the successful coordination, however. The answers to this question are pertinent to both CSCW and software engineering research in general.

2. How can open-source development achieve agreement among members and consistency in design using limited media?

Keeping consistency in design and reaching agreement among members are difficult among weakly tied collaborators. Small problems of this kind can become far bigger as a project progresses. Open-source development usually manages to avoid the problem.

3. How can open-source development achieve continuous innovations using limited media?

To set out, elaborate and implement innovations, tacit knowledge such as skill and intuition must be shared [18]. Informal communication, which is resourceful for problem solving and essential for innovation, is important [14]. All of these requisites cannot be provided if computers mediate the communication.

METHOD

The principal method used in this study was observation. All the communication that took place in development was observed as virtually all messages were posted to the mailing lists. Interviews with members revealed that personal exchange among members was rare (all said almost 100% were sent to the lists) although a broader survey would be necessary to confirm this is the norm. From the observations, the development practices were broadly identified and ranged from the division of labor to the roles of leadership.

Quantitative methods were employed to illustrate the communication patterns. In order to figure out communication patterns, the content analysis method proposed by Krippendorff [16] was adopted and the replicability of the analysis was guaranteed by computing the agreement rates, Cohen's κ. Tasks were identified and counted to show the process of individual work. These quantitative data sets are to illustrate rather than corroborate observations.

Interviews were conducted with 10 members of one of the projects. These interviews were organized by the standard process of ethnographic interviews [23]. Programming vocabulary was used in the interviews. Questions started with the simple and progressed to the more complex. These questions ranged from "How do you start work?" to "How do you manage the source code?" Ad hoc sub-questions were posed depending on the responses.

CASES

Two open-source projects were selected as cases.

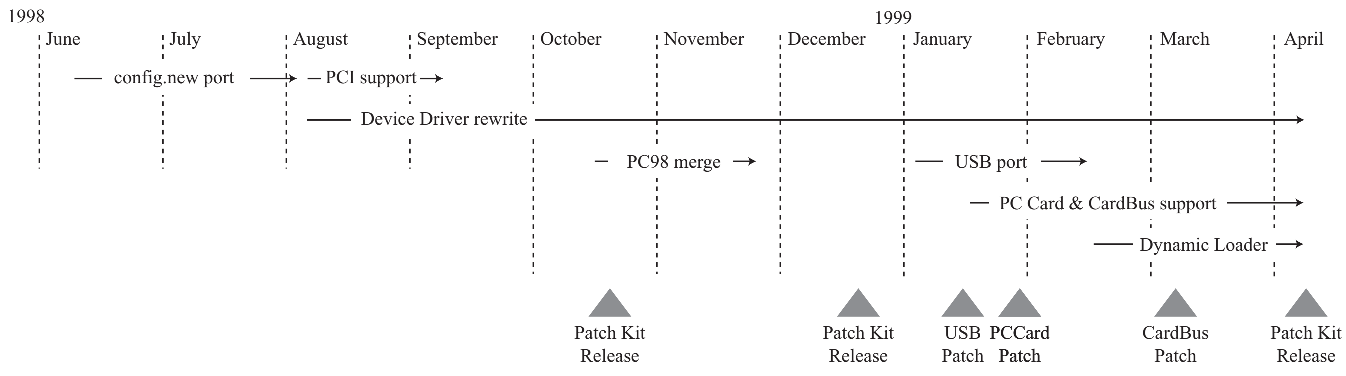

Figure 1: The Development Path in Newconfig Project

1. FreeBSD Newconfig Project, was relatively small and aimed at developing a new program.
2. GNU GCC Project, primarily maintained existing programs.

Case 1: FreeBSD Newconfig Project[1]

Overview

FreeBSD is a free UNIX operating system inheriting the code from 4.4BSD-Lite, the last release from University of California, Berkeley. FreeBSD is oriented to the focused functionality and the ease of installation primarily on PCs. NetBSD and OpenBSD, other BSD derivatives, are ported to multiple platforms. FreeBSD is widely used as both server and client systems by home users, research institutions, public sectors and companies.

The device configuration of FreeBSD had serious flaws since it was based on the obsolete ISA architecture. In order to implement PCI and other new devices, the modification of the part was urgent. This frustration was perceived among the developers of PCCard support for FreeBSD, a Japanese organized project. Those members gathered to embark on the new project with the objective to implement 'config.new', already working in other BSDs.

The project was chosen for the research study on the criteria that the project was attempting to create a new software program. We expected to observe the collaborative development processes in detail rather than the mere maintenance work. This project had a high level of uncertainty as to whether the resulting work would be accepted by other FreeBSD developers as the work was supposed to alter dramatically the parts on which other projects depended. Yet the project required major work input since once the device configuration part was updated, all the device drivers would have to be modified.

Through the research project, 1,838 messages were observed between June 1998 (the commencement of the project) and April 1999 (the release of an almost complete product). The research project was later augmented by interviews with ten members of the team. The project failed to accomplish the initial goal although the program was completed and released. The core members of the FreeBSD community did not accept the results and chose another project that paralleled Newconfig Project[2]. The resulting code was transferred to other projects such as NetBSD and several developers continue the work.

Development Process

The study found that the program was constructed by repeatedly adding components to the core structure as shown in Figure 1. The core mechanism of the device configuration, config.new, was ported from NetBSD. Subsequently, PCI support was implemented on the config.new and device drivers were gradually rewritten to comply with the new mechanism. At appropriate times, in October and December, patch kits to install the newconfig-based FreeBSD were released. After stabilizing the program, the project started to implement such new functions as USB support, PCCard and CardBus support and dynamic loader. The dynamic loader, which enabled flexible manipulation of devices after booting, was incorporated since FreeBSD community had a consensus that dynamic loading would replace the static one.

The detailed development process defined through observation is shown in Figure 2. The developers began by discussing the specifications of the product, such as behaviors of the control mechanism, data structures, potential enhancements, and the development milestones. This discussion, however, contained rough idea rather than detailed documented structure for the product. The main objective was to share and merge various ideas of the members. The discussion resulted in a TODO list, a task list. Programmers selected an item from the list and implemented it. Somewhat surprisingly, they did not usually declare the commitment to the task before starting. After finishing the task, they submitted a patch to the project's mailing list. Another member who had time and knowledge reviewed it by reading the code. On the OK'ing by the reviewer, the work was included in the shared code repository. Official releases

[1] See http://www.jp.freebsd.org/newconfig

[2] Although the failure is beyond the scope of this study, an inquiry is necessary to broadly understand the open-source software.

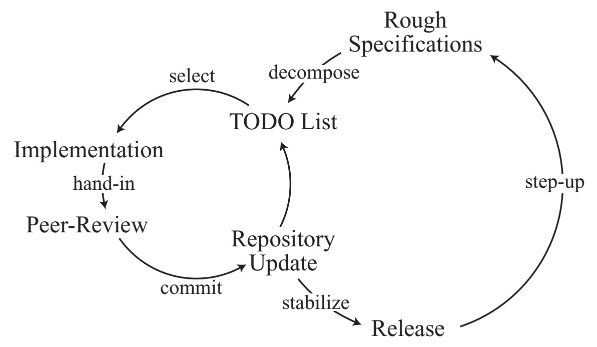

Figure 2: Development Process of Newconfig Project

were announced when the product became stable and provided enough functions.

In this process, TODO lists played a crucial role. To facilitate the efficient work, the form of TODO lists was highly elaborated. First, each item representing each task had an indication of priority level scored from A+ to C- with A+ the most urgent. Second, each item also had a difficulty level from A+ to C-. Finally, a short description was attached to an item. Importantly, the length of the descriptions was limited up to two lines. This means that the descriptions were not to specify the content of the task but to explain the type of the task. The way of performing tasks was decided by the individual team members.

During the development, the project maintained one unified set of source files in its source repository of CVS. The project had started with setting up a CVS repository and the compilation of mailing lists. Each task began with copying the master code of the repository into each individual's workspace. The programmers modified the code and tested the modification in the local copy. When finishing the implementation, they extracted the difference between the modified version and the central master code with 'diff' and then submitted the difference to a mailing list.

Another important aspect was that this project had grown during the development. At the outset, only about ten people had gathered to start the project but finally 54 members were involved. The openness would appear key for successful development.

Communication Pattern

Communication among members averaged 4.29 messages per day. To understand the communication pattern, each of 479 sampled messages was coded into one of the five categories, 'question', 'response', 'hand-in', 'proposal' and 'other' employing the content analysis method [16]. The coding criteria and frequencies are shown in Table 1. 'Question' is any message to ask for information both about technical and managerial matters. 'Response' includes any replies to previous messages from answers for questions, agreement, disagreement and additional information. Although the study initially divided this category to more fine-grained ones, we put all into 'response' since we could not achieve reliable agreement rates. 'Hand-in' is a general report of any kind of action, for example, patches, bug reports and successful test results. Finally, 'proposal' is the message that contains ideas for improvement or problem solving. The reliability measure is the agreement rate between two independent trained coders.

Table 1: Message Categories in Newconfig

Category	Coding Criteria	Frequency %	Reliability Cohen's κ
Question	Asking for information	18.8	.898
Response	Responding to messages	49.9	.859
Proposal	Proposing an idea	9.2	.839
Hand-in	Reporting results of action	14.2	.960
Other	Residual category	7.9	.817

N=479

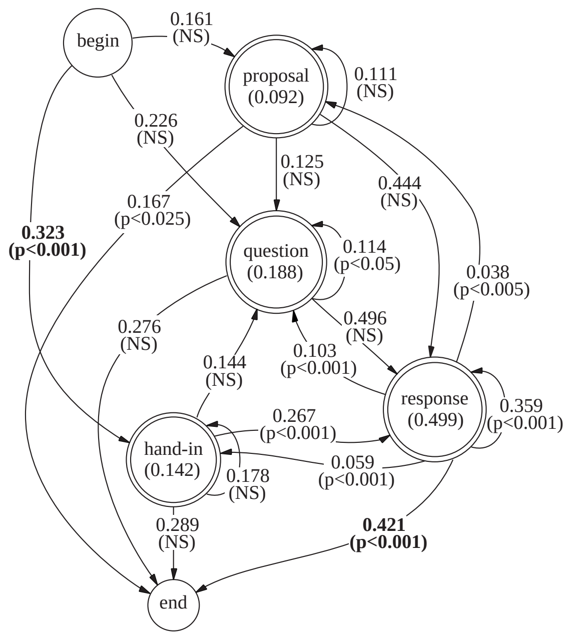

Figure 3: Communication Pattern of Newconfig

We present the communication pattern as a transition diagram of message categories in Figure 3[3]. We separated the series of messages according to discussion threads indicated by the references in 'References' and/or 'In-Reply-To' fields of the electronic mail standard. The 'begin' and 'end' labels indicate the beginning and the end of threads. The value under a category label is the occurrence probability of the category and the value attached to an arc is the transition probability. Small probabilities (occurrence probability of category a x transition probability from a to $b < 0.01$) and 'other' are eliminated. P-values and NS (not significant) in the figure indicate the significance levels of χ^2 test against the null

[3] This figure was constructed by an open-source program DOT, distributed by AT&T, with slight manual modification.

hypothesis that the transition is random (transition probability from *a* to *b* = occurrence probability of *b*).

Significant transitions (p<.001) are marked with a bold font. Since the significance was calculated with two-tailed χ^2 test, the lower rejection region was ignored. The large transition probability from 'begin' to 'hand-in' means that discussion took place after the report of action. This fact is at variance with the traditional view that action should follow discussion and planning.

Case 2: GNU GCC Project[4]

Overview

GNU Compiler Collection (GCC) Project, one of the most renowned projects of GNU, has its origin in the C compiler Richard Stallman created to make free UNIX. Now, the project maintains various compilers and libraries including C++, Pascal and Fortran although it was initially called GNU C Compiler (GCC) Project. The high performance and stability result in the compilers being widely used to create programs.

GCC has experienced two major discontinuities in its long history. The first was the leap to version 2, GCC2, in 1991. Since the original C compiler turned out to have fundamental limitations in design, it had to be dramatically re-implemented. The second was due to the dilemma that the work of many programmers could not be incorporated in the main trunk of the code because of the project's orientation to stability. In April 1997, a new project, Experimental GNU Compiler System (EGCS), was founded to overcome this dilemma by employing open-style development. The EGCS Project was appointed as the official maintainer of the GNU compilers in 1998 and renamed as GCC.

This project was selected for the study in order to analyze the advertised advantages of open-source (i.e., "Given enough eyeballs, all bugs are shallow [20]."). Although the Newconfig Project demonstrated an open-style development, it was rather small and most members were Japanese. Although the investigation of GCC is still in progress, it can serve as the support for the findings. While observations of the total of 1,012 messages are complete, it was not possible to augment these observations with interview since the members of GCC are dispersed worldwide.

The steering committees consist of 13 members, each of whom represents a certain community, for example, the academic and Fortran users. They have slightly large influence in decision amking. In addition, 134 programmers and 35 testers joined the development. The list of the contributors is presented on a web page as a reward incentive.

Development Process

The study shows the development process of GCC to be straightforward since it involved maintenance of software and not software creation. While there were some special

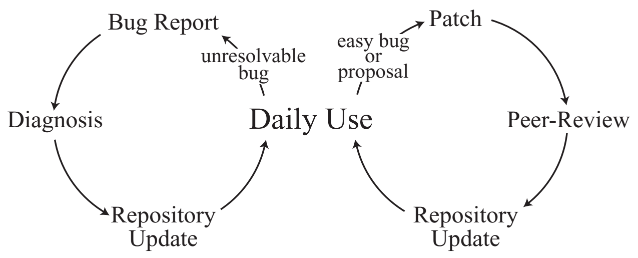

Figure 4: Development Process of GCC Project

elements to the project, for example a subproject of new instruction scheduling support, most work originated from daily use of the program as shown in Figure 4. This daily use afforded the opportunities to discover bugs and develop fixes. Bug reports entailed diagnosis and further fix and resulting patches were reviewed by a peer and submitted to the central source repository. In case of small bugs or proposals, users sent patches after fixing the bugs or implementing the proposals. Peer-review followed the patches in the same fashion as Newconfig.

Bugs had to be reported carefully according to the 'bug reporting guideline' presented on the web. This guideline requested users to refer to manuals, FAQ and a list of known bugs before reporting to the list. Bug reports needed to include the GCC version, the system type, options passed to the compiler, and output of the preprocessor. These instructions were aimed to minimize replication and to ensure that sufficient information to detect bugs was provided.

Communication Pattern

There was much more communication than in Newconfig with an average of 37.1 messages per day. In exactly the same fashion as Newconfig, 552 messages were coded and a transition diagram of message categories was constructed as shown in Table 2 and Figure 5. The high transition probability from 'begin' to 'hand-in' in this figure shows the same tendency as Newconfig that communications began with the report of action rather than the planning before action. The significant transition from 'hand-in' to 'end' means that the report of successful test results and trivial actions did not require replies. The high occurrence probability of 'hand-in', compared with that in Newconfig, suggests that there was less communication on product design and more action such as improvement, debugging and enhancement since the project focused on maintaining existing software, and not creating it.

FINDINGS

Bias for Action

Open-source programmers are biased towards action rather than coordination. They tend to act without first declaring the commitment. As shown in Figure 3 and Figure 5, the report of action triggers discussion more frequently than discussion initiates action. In another analysis, 70.8% and 80.3% of tasks we identified were started without declaration and 54.9% and 60.7% were finished entirely without any

[4] See http://www.gnu.org/software/gcc/gcc.html

Table 2: Message Categories in GCC

Category	Coding Criteria	Frequency %	Cohen's κ
Question	Asking for information	12.0	.823
Response	Giving information	43.8	.931
Proposal	Proposing an idea	5.8	.889
Hand-in	Reporting results of action	34.4	.945
Other	Residual category	4.0	.948

N=552

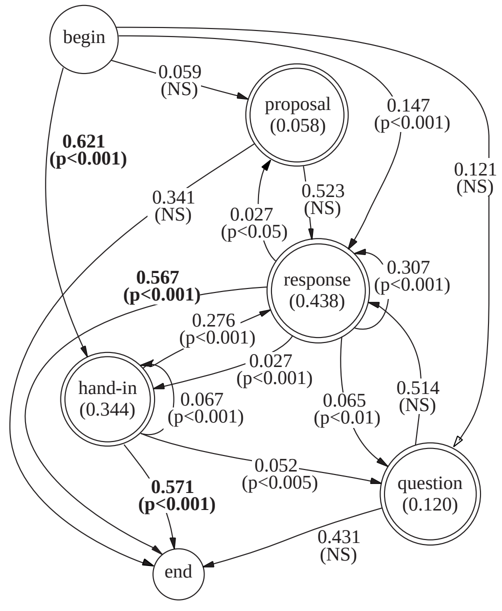

Figure 5: Communication Pattern of GCC

notification of the commitment in Newconfig (N=71) and GCC (N=56) respectively. Since electronic media make it difficult to discuss the plan before action, it is much easier to show the results after action.

This bias facilitates experiments where the expected outcome is uncertain and which would be discouraged in discussions before action. One member said, "I'm always relaxed. I can fail because I don't tell others what I do.[5]" These hidden experiments are the important source for innovations. The bias also makes it easy for potential developers to join the projects. Potential members who do not have confidence in their skill and knowledge are discouraged if they have to declare before joining projects. Moreover, the bias helps avoid procrastination. It is likely that too much discussion inhibits development.

The study also found that coordination followed action. In both cases, all the work required 'peer-review' after results were posted. Through the peer-review, flaws were detected and improvements were suggested. This is an important process to maintain the consistency in implementation as well as to guarantee the stability. Reflection after action is an important occasion for innovations as well [27].

Along with the bias for action, the work of programmers is loosely coupled. 79.7% and 90.4% of tasks were completed by one person with no information shared in Newconfig and GCC respectively. Olson and Teasley also observed that since computers could not support the tight coupling, an organization of work was altered so that members would not need tight coupling [19]. One member explained the difficulty of sharing work,

> [If mediated by computers,] in case of something easy, the same thing happens to some degree. But in case of complex things, it is likely that the ideas of many people lead to the chaos. No consistency, for example.

Rational Culture

The culture of open-source communities can be characterized by the orientation to a rational rather than lateral approach. Members try to make their behavior logically plausible and technologically superior options are always chosen in decision-making. We use the term 'culture' since the orientation to rationality is taken for granted and implicitly shapes the behavior of project members. This rational culture is necessary to communicate through computers, which make agreement difficult [11, 12]. Rationality is the only criterion by which everyone can agree to decisions. One informant said,

> …That is, we don't meet face-to-face. Then, we need some criteria to decide something, right? The criterion that everyone understands is finally only 'technologically good or bad.'

The characteristics of electronic media, in turn, favor the rational decision-making. The reduction of social context cues by electronic media equalizes participants [24]: Even a CEO and a high school student can have the same presence. The authority does not work in this situation. In addition, computer-mediated communication is inherently impersonal and prompts task-oriented and focused exchanges [11]. Accordingly emotional or authoritative factors are precluded in the communication.

Furthermore, the asynchronous nature of electronic mail gives members time to reflect on messages they wrote and render the messages logical before sending them. In this process, they prepare plausible reasoning, ponder on alternatives and remove superfluous information. Programmers also refer to such resources as code files, web pages, research papers and messages of others to make the idea clear and reliable. By reflecting before sending messages, members not only make their messages plausible but also learn and improve their skill and knowledge by ad hoc learning. One member said in an interview,

> On average, I consider for 30 minutes, [and] sometimes one hour to send a message for Newconfig… [T]he issues are difficult technologically… If I don't consider deeply, the information will be useless. I construct my opinions strictly... I make it clear that the information is based on something. [For example] "I came up with this idea by reading this part of the source code" or "I think so because I heard someone's opinion."

[5] All the transcripts of interviews were translated from Japanese by the authors. The authors interpolated words in brackets.

Written communication by electronic mail also helps break down vague ideas and serves as a reliable organizational memory. A team member said,

> It is difficult to orchestrate work only with vague images [discussed face-to-face]. The information about who does what and why should be written down and accessible to anyone later on [by electronic mail].

In addition, the organizational culture favors results of action rather than abstract discussions. Even if one has a potentially excellent idea, it cannot be accepted without source code realizing the idea. This analysis explains why Linux, which was technologically unsurprising, became more widespread than GNU Hurd, which was advertised but late to be released. Without the source code, dispersed developers cannot collaborate. This result-based culture fosters bias for action and avoids procrastination.

Even if decisions are made rationally, leaders play a significant role in many aspects of development. Importantly, they are recognized as leaders because they can produce technologically outstanding programs and achieve results. These leaders are not necessarily appointed but emergent. Their role is to expedite decisions and keep the project moving forward. In open-source initiatives, only a good programmer who can critique and understand the design can be a leader.

Effective Media Use: CVS, TODO Lists and Mailing Lists

Maintaining project consistency is difficult if computers mediate communication. Loose team structures seem to aggravate the problem. Yet, both of the project studied demonstrated the effective media use to avoid the problem.

CVS

CVS (Concurrent Versions System) plays a significant role in open-source software development. CVS typically operates in the following way. First, developers copy (using a command 'checkout') entire source code from the central repository. After modifying the code, they update their local copy by comparing with the central code. If the central source code is changed while they are modifying their local copy, conflict occurs. If these parallel changes have no relation with each other (e.g., in different files or in different parts in the same file), the conflict can be automatically resolved by CVS. Otherwise, developers need to resolve it manually. By a command 'commit', the local copy can be updated to the repository.

The primary role of CVS is to centralize the source code so that developers can always refer to the latest code. This centralization gives consistency in development organizations. One project member said,

> In 386BSD [another open-source], many made patches without any policy and sent them to mailing lists. Because there was no person who managed a source repository, patches couldn't be organized… The person who cares for the integrated source code is important in software development.

Maintaining source code that works anytime is a crucial element merging the work of dispersed individuals. This is particularly the case for large-scale software development, which comprises large teams, since it is hard to keep the consistency across organizational boundaries [5]. In this sense, CVS repositories are 'boundary objects', artifacts that are shared across boundaries and provide shared context even if individuals focus on distinct aspects of the objects [25]. Boundary objects provide backbones of practices among loosely coupled individuals.

The most important advantage of CVS is that developers can always keep the whole code in their local workspace. All members can compile, test and use the program at the same time. Developers do not need coordination by locking files before starting work and therefore work can occur spontaneously. CVS is a good balance between centralization and spontaneous work.

It is also worth noting that CVS fosters the competition among developers. Developers explicitly rush and try to complete work as quickly as possible. It is often claimed that quality is sacrificed when speed is favored. In open-source software, however, development style is different from traditional approaches. Software is not implemented as formally designed but refined repeatedly with the involvement of many workers. For this iterative development, the program stored in a CVS repository is required to work anytime even if it is not in perfect quality. Quality is guaranteed through the iteration.

TODO Lists

TODO lists, widely used in open-source projects, are also a centralization effort to manage dispersed work. TODO lists provide a rough idea about what to do next. On the other hand, TODO lists are important for spontaneous work as well. As depicted in Figure 2, work starts spontaneously from TODO lists in Newconfig. By referring to a TODO list, developers do not usually declare the commitment to the task they choose.

TODO lists do not specify the details of tasks but provide the whole map of the work to be done. Programmers are endowed with the right to decide the details of implementation as they want to. Since any work is a situated action and a plan cannot prescribe all the contingencies in advance, this loose organizing is a moderate choice in particularly complex software development [26]. In the notion of Schmidt [21], TODO lists serve as 'maps', which not determine but orient actions, rather than 'scripts', which coercively determine actions. Furthermore, this approach, giving individuals enough room to innovate, is essential to produce innovative as well as stable products [5]. Espoused descriptions of work often differ from the actual processes and therefore the focus on them impedes learning and innovation [1].

Mailing Lists

Interviews with team members identified the intensive use of a mailing list. Although we predicted that lateral communication would be frequent and produce information asymmetry, project members usually post all messages to the mailing list. Broadcasting all communications makes the development work transparent. This transparency provides, in a limited way, one level of awareness about what others are doing. This is similar to 'over-hearing' a conversation between others or people 'talking out loud' about their own work [10]. Through overhearing, members can perceive what is going on. One informant put it,

> When I skim through the mailing lists, I can know mostly what others are involved with. Well, sometimes I cannot understand difficult issues but at least can perceive what is going on.

Information overload due to broadcast communication does not pose a problem. As mentioned earlier, project members usually prepare messages carefully and sometimes do not send them. Members, in turn, just look through or over-hear messages and read only messages of their interest carefully. In addition, it is required to consult FAQ and a list of known bugs on the web before sending bug reports to minimize the traffic.

IMPLICATIONS

For over a decade, CSCW researchers have been challenging the limitations of electronic media by proposing advanced technologies. Some have tried to provide users with awareness of what is going on around them to initiate informal and rich conversations (e.g., [7]). Others have proposed multimedia conferencing systems to enable as rich interactions as face-to-face (e.g., [17]). Unlike those, this study sheds light on organizing practices as well as technological advances to challenge traditional limitations. Few in the CSCW field have paid attention to the organizational reactions to lean media although it is of grave importance to not only managers who introduce those systems but also the designers of collaborative systems.

The primary contention of this study is that dispersed collaborations require not only centralization and transparency of work but also spontaneous work and bias for action. Both traditional technology support (e.g., workflow management systems) and management practice (e.g., hierarchical control) are appropriate to make dispersed work converge but oblivious to spontaneous work and thus emergent innovations. In case of dispersed collaboration, these traditional approaches are doomed to fail.

For implications to technology designers, CVS and less specified TODO lists are in a good balance between these conflicting requirements. For managers or individual workers in dispersed organizations, the relationship between action and coordination should be inversed and too much coordination before action must be avoided. Bias towards action needs to be viewed as an essential source of innovation.

It is our second claim that in dispersed collaborations organizational culture should be nurtured so that team members act rationally and rational options are chosen in decision-making. Rationality is the only criterion by which people agree when computers limit communication. If members cannot agree to decisions, they lose interest and motivation and thus spontaneous work would not emerge.

CONCLUSION

In this paper, we presented the findings that address the research question, "How can open-source software development achieve smooth coordination, consistency in design and agreement among members, and continuous innovations while mediated by computers?" The findings suggest that spontaneous work coordinated afterward is effective and rational culture helps achieve agreement among members. Communications media (CVS, TODO lists and Mailing lists) are used in a good balance between centralization and spontaneity.

This study implies that a traditional approach—coordination precedes action—is not appropriate in dispersed collaborations. Instead, system designers and managers should pay attention to spontaneous work. Technological support should not only centralize work but also gives individuals room to act locally and innovate. In addition, rational organizational culture needs to accompany in order to make the development consistent and merge spontaneous work smoothly.

A next step would be to generalize the open-source approach to ordinary organizations. Geographically dispersed organizations are now recognized as a source of innovation: Consultants work in clients' sites, sales staff access customers directly from their home office, and knowledge learned locally can be distributed both nationally and internationally. The effective practices and media use of open-source, which were revealed, are meaningful in these situations.

ACKNOWLEDGMENTS

We thank all the members of two open-source projects, Newconfig and GCC for their cooperation. Prof. Toshihiro Kanai of Kobe University has supported this study from the outset. We are also grateful to Prof. Ikujiro Nonaka of Hitotsubashi University, Pól Ó Móráin of Xerox Venture Lab Europe and anonymous reviewers for their advice.

REFERENCES

1. Brown, J. S. and Duguid, P. Organizational Learning and Communities-of-Practice: Toward a Unified View of Working, Learning, and Innovation, *Organization Science*, 2(1), 1991, 40-57
2. Button, G. and Sharrock, W. Project Work: The Organisation of Collaborative Design and Development in Software Engineering, *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 5, 1996, 369-386
3. Cederqvist, P. et al. *Version Management with CVS*, Linkoping, Sweden: Signum Support AB, 1993
4. Curtis, B., Krasner, H. and Iscoe, N. A Field Study of the Software Design Process for Large Systems, *Communications of the ACM*, 31(11), 1988, 1268-1287

5. Cusumano, M. A. and Selby, R. W. How Microsoft Builds Software, *Communications of the ACM*, 40(6), 1997, 53-61

6. Daft, R. L. and Lengel, R. H. Organizational Information Requirements, Media Richness and Structural Design, *Management Science*, 32(5), 1986, 554-571

7. Dourish, P. and Bly, S. Portholes: Supporting Awareness in a Distributed Work Group, In *Proceedings of ACM Conference on Human Factors in Computing Systems CHI'92*, 1992, 541-547

8. Grinter, R. E. Supporting Articulation Work Using Software Configuration Management Systems, *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, 5, 1996, 447-465

9. Grinter, R. E. Recomposition: Putting It All Back Together Again, In *Proceedings of ACM Conference on Computer Supported Cooperative Work CSCW'98*, 1998, 393-402

10. Heath, C. and Luff, P. Collaboration and Control: Crisis Management and Multimedia Technology in London Underground Line Control Rooms, *Computer Supported Cooperative Work*, 1, 1992, 69-94

11. Hiltz, S. R. and Turoff, M. *The Network Nation*, Cambridge MA: MIT Press, 1993

12. Kiesler, S., Siegel, J. and McGuire, T. W. Social Psychological Aspects of Computer-Mediated Communication, *American Psychologist*, 39(10), 1984, 1123-1134

13. Kiesler, S., Zubrow, D., Moses, A. M. and Geller, V. Affect in Computer-Mediated Communication: An Experiment in Synchronous Terminal-to-Terminal Discussion, *Human-Computer Interaction*, 1(1), 1985, 77-104

14. Kraut, R. E., Fish, R. S., Root, R. W. and Chalfonte, B. L. Informal Communication in Organizations: Form, Function and Technology, In Oskamp, S. and Scacapan, S. (Eds.) *Human Reactions to Technology: Claremont Symposium on Applied Social Psychology*, Beverly Hills, CA: Sage Publications, 1990

15. Kraut, R. E. and Streeter, L. Coordination in Software Development, *Communications of the ACM*, 38(3), 1995, 69-81

16. Krippendorff, K. *Content Analysis: An Introduction to Its Methodology*, Newbury Park, CA: Sage Publications, 1980

17. Nakanishi, H., Yoshida, C., Nishimura, T. and Ishida, T. FreeWalk: Supporting Casual Meetings in a Network, In *Proceedings of ACM Conference on Computer Supported Cooperative Work CSCW'96*, 1996, 308-314

18. Nonaka, I. and Takeuchi, H. *The Knowledge Creating Company*, New York, NY: Oxford University Press, 1995

19. Olson, J. S. and Teasley, S. Groupware in the Wild: Lessons Learned from a Year of Virtual Collocation, In *Proceedings of ACM Conference on Computer Supported Cooperative Work CSCW'96*, 1996, 419-427

20. Raymond, E. S. *The Cathedral and the Bazaar*, Sebastopol, CA: O'Reilly & Associates, 1999

21. Schmidt, K. Of Maps and Scripts—Status of Informal Constructs in Cooperative Work, In *Proceedings of ACM Conference on Supporting Group Work Group'97*, 1997, 138-147

22. Short, J., Williams, E., and Christie, B. *The Social Psychology of Telecommunications*, London: John Wiley & Sons, 1976

23. Spradley, J. P. *The Ethnographic Interview*, New York, NY: Holt, Rinehart and Winston, 1979

24. Sproull, L. S. and Kiesler, S. Reducing Social Context Cues: Electronic Mail in Organizational Communication, *Management Science*, 32(11), 1986, 1492-1512

25. Star, S. L. The Structure of Ill-Structured Solutions: Boundary Objects and Heterogeneous Distributed Problem Solving, In Gasser, L. and Huhns, M. (Eds.) *Distributed Artificial Intelligence Vol. 2*, San Mateo CA: Morgan Kaufmann, 1989, 37-54

26. Suchman, L. A. *Plans and Situated Actions: The Problem of Human-Machine Communication*, New York, NY: Cambridge University Press, 1987

27. Weick, K. E. *Sensemaking in Organizations*, Thousand Oaks, CA: Sage Publications, 1995

28. Wenger, E. *Communities of Practice: Learning, Meaning, and Identity*, New York, NY: Cambridge University Press, 1998