Coordination of Free/Libre Open Source Software Development

Kevin Crowston[*] Kangning Wei[†] Qing Li[‡]
U. Eseryel[**] James Howison[††]

[*]Syracuse University School of Information Studies
[†]Syracuse University School of Information Studies
[‡]Syracuse University School of Information Studies
[**]Syracuse University School of Information Studies
[††]Syracuse University School of Information Studies

COORDINATION OF FREE/LIBRE OPEN SOURCE SOFTWARE DEVELOPMENT

Kevin Crowston, Kangning Wei, Qing Li,
U. Yeliz Eseryel, and James Howison
Syracuse University School of Information Studies
Syracuse, NY U.S.A.
crowston@syr.edu **kwei@syr.edu** **qli03@syr.edu**
uyeserye@syr.edu **jhowison@syr.edu**

Abstract

The apparent success of free/libre open source software (FLOSS) development projects such as Linux, Apache, and many others has raised the question, what lessons from FLOSS development can be transferred to mainstream software development? In this paper, we use coordination theory to analyze coordination mechanisms in FLOSS development and compare our analysis with existing literature on coordination in proprietary software development. We examined developer interaction data from three active and successful FLOSS projects and used content analysis to identify the coordination mechanisms used by the participants. We found that there were similarities between the FLOSS groups and the reported practices of the proprietary project in the coordination mechanisms used to manage task-task dependencies. However, we found clear differences in the coordination mechanisms used to manage task-actor dependencies. While published descriptions of proprietary software development involved an elaborate system to locate the developer who owned the relevant piece of code, we found that "self-assignment" was the most common mechanism across three FLOSS projects. This coordination mechanism is consistent with expectations for distributed and largely volunteer teams. We conclude by discussing whether these emergent practices can be usefully transferred to mainstream practice and indicating directions for future research.

Keywords: Free/libre open source software development, coordination theory, coordination mechanisms

Introduction

The apparent success of free/libre open source software[1] (FLOSS) development projects such as Linux, Apache, and many others has raised the question, what lessons from FLOSS development can be transferred to mainstream software development? In this paper, we address this question for one key problem in large-scale software development, namely coordination of the work of multiple developers.

Coordination is a significant concern in large software development teams. Curtis et al. (1988) found that large projects have extensive communication and coordination needs that are not mitigated by documentation. These coordination problems are exacerbated when the development team is geographically or organizationally distributed, as is increasingly common. More effort

[1]The free software movement and the open source movement are distinct and have different philosophies but mostly common practices. The licenses they use allow users to obtain and distribute the software's original source code, to redistribute the software, and to publish modified versions as source code and in executable form. While the open source movement views these freedoms pragmatically (as a development methodology), the free software movement regards them as human rights, a meaning captured by the French/Spanish word *libre* and by the saying "think of free speech, not free beer." (See http://www.gnu.org/philosophy/ and http://opensource.org for more details.) This paper focuses on the development practices of these teams, which are largely shared across both movements. However, in recognition of these two communities, we use the acronym FLOSS, standing for Free/Libre and Open Source Software, rather than the more common OSS.

is required for interaction when participants are distant and unfamiliar with each others' work (Ocker and Fjermestad 2000; Seaman and Basili 1997). Curtis et al. noted that coordination breakdowns were likely to occur at organizational boundaries, but that coordination across these boundaries was often extremely important to the success of the project. The additional effort required for distributed work often translates into delays in software release compared to traditional face-to-face teams (Herbsleb et al. 2001; Mockus et al. 2002). The problems facing distributed software development teams are reflected in Conway's law, which states that the structure of a product mirrors the structure of the organization that creates it. Accordingly, splitting software development across a distributed team will make it hard to achieve an integrated product (Herbsleb and Grinter 1999). Despite these challenges, some FLOSS development teams have been quite successful in overcoming the challenges of distributed software development. In this paper, we analyze the coordination mechanisms used in FLOSS development to answer our research question, namely, how coordination of FLOSS differs from proprietary software development, and to explore what proprietary development might learn from FLOSS about coordinating the work of distributed developers.

Coordination of developers working on proprietary software has been extensively studied. Curtis et al. studied how requirement and design decisions were made, represented, and communicated, and how these decisions impacted subsequent development processes for large systems. Kraut and Streeter (1995) found that the formal and informal communication mechanisms for coordinating work on large-scale, complex software projects were important for sharing information and achieving coordination in software development. Crowston (1997) used coordination theory to analyze coordination mechanisms in software change processes, including task assignment, resource sharing, and managing dependencies between modules of source code, in order to suggest alternative processes. Kraut et al. (1999) found that reliance on personal linkages rather than electronic networks contributed to coordination success. Faraj and Sproull (2000) found a strong relationship between expertise coordination and team performance, over and above the contribution of team input characteristics, presence of expertise, and administrative coordination. Espinosa et al. (2001) tested whether implicit mechanisms such as shared mental models are used for coordination for geographically distributed projects. In a later study, Espinosa et al. (2004) concluded that an effective strategy for coordination success involves finding a mix of explicit and implicit coordination mechanisms appropriate for the task, which may change as the task progresses across time.

However, the archetypical community-based FLOSS development process differs greatly from proprietary development in several respects that may affect or depend on the coordination mechanisms used. (In focusing on community-based development, we exclude projects such as MySQL that are developed by a single organization following a conventional software engineering approach and only released under a FLOSS license.) A primary difference is that the community-based development process is not owned by a single organization. Developers contribute from around the world, meet face-to-face infrequently if at all, and coordinate their activity primarily by means of computer-mediated communications (CMC) (Raymond 1998; Wayner 2000) and other software development tools (e.g., source code control systems). As noted above, the research literature emphasizes the difficulties of distributed software development, but the case of FLOSS development presents an intriguing counter-example.

While distributed work is becoming more common even for proprietary software development, what is perhaps most surprising about the FLOSS process is that it appears to eschew traditional project coordination mechanisms such as formal planning, system-level design, schedules, and defined development processes (Herbsleb and Grinter 1999). As well, many (though by no means all) programmers contribute to projects as volunteers, without working for a common organization or being paid. Generally, a small core group oversees the overall design and contributes the bulk of the code, but other developers play an important role by contributing bug fixes, new features, or documentation, by providing support for new users, and by filling other roles in the teams. Characterized by a globally distributed developer force and a rapid and reliable software development process, effective FLOSS development teams somehow profit from the advantages and overcome the challenges of distributed work (Alho and Sulonen 1998), making their practices potentially of great interest to mainstream development.

In the next section, we briefly review coordination theory and show how it can guide the analysis of a process such as software development. The data and method section describes the FLOSS projects we analyzed, our data elicitation and analysis approach, and the rationale for their selection. The results section presents the dependencies and coordination mechanisms identified in the cases. We conclude with some observations about the coordination of software development and suggestions for further research.

Theory: A Coordination Theory Approach to Organizational Processes

To analyze coordination in FLOSS development processes, we use the framework developed by Malone and Crowston (1994), who define coordination as "managing dependencies between activities" (p. 90). They define coordination theory as the still developing body of "theories about how coordination can occur in diverse kinds of systems" (p. 87). Coordination theory analyzes group action in terms of *actors* performing *interdependent tasks* that may also require or create *resources* of various types. For example, in the case of software bug fixing, *tasks* include diagnosing the bug, writing code for a fix and integrating it with the rest of the system, as mentioned above. *Actors* include the customers and various employees of the software company. In some cases, it may be useful to analyze a group of individuals as a collective actor (Abell 1987). For example, to simplify the analysis of coordination within a particular subunit, the other subunits with which it interacts might all be represented as collective actors. Finally, *resources* include the problem reports, information about known problems, computer time, software patches, source code, and so on.

It should be noted that in developing this framework, Malone and Crowston describe coordination mechanisms as relying on other necessary group functions, such as decision making, communications and development of shared understandings, and collective sense-making (Britton et al. 2000; Crowston and Kammerer 1998). To develop a complete model of some process would involve modeling all of these aspects: coordination, decision-making, and communications. In this paper, our analysis will focus on the coordination aspects, bracketing the other phenomenon.

According to coordination theory, actors in organizations face *coordination problems* that arise from dependencies that constrain how tasks can be performed. These dependencies may be inherent in the structure of the problem (e.g., components of a system may interact with each other, constraining the kinds of changes that can be made to a single component without interfering with the functioning of others) or they may result from decomposition of the goal into activities or the assignment of activities to actors and resources (e.g., two engineers working on the same component face constraints on the kind of changes they can make without interfering with each other).

To overcome these coordination problems, actors must perform additional activities, what Malone and Crowston call *coordination mechanisms*. For example, a software engineer planning to change one module in a computer system must first check if the changes will affect other modules and then arrange for any necessary changes to modules that will be affected; two engineers working on the same module must each be careful not to overwrite the other's changes. Coordination mechanisms may be specific to a particular setting, such as a code management system to control changes to software, or general, such as hierarchical or market mechanisms to manage assignment of activities to actors or other resources.

There are often several coordination mechanisms that could be used to manage a dependency, as the task assignment example illustrates. Possible mechanisms to manage the dependency between an activity and an actor include a manager selection of a subordinate, first-come-first-served allocation and various kinds of markets. Again, coordination theory suggests that these mechanisms may be useful in a wide variety of organizational settings. Organizations with similar activities to achieve similar goals will have to manage the same dependencies, but may choose different coordination mechanisms, thus resulting in different processes. In this paper, we will consider if the coordination mechanisms used in FLOSS development differ from those described in the literature for proprietary software development, and if so, what mechanisms might be transferred.

As a guide to applying coordination theory, Crowston (2003) presents a typology of dependencies and associated coordination mechanisms. The main dimension of the typology involves the types of objects involved in the dependency. To simplify the typology, we compress the elements of Malone and Crowston's framework into two groups: tasks and resources used or created by tasks (which here includes the effort of the actors). Logically, there are three kinds of dependencies between tasks and resources: those between two tasks, those between two resources, and those between a task and a resource. Each of these dependencies has an associated set of appropriate coordination mechanisms.

Methods: Inductive Coding of FLOSS Developer E-Mail Interactions

In this paper, we use coordination theory to analyze FLOSS software development processes in order to compare the mechanisms used in FLOSS to those described in the literature for proprietary development. Because the coordination mechanisms used by these teams have not yet been described, we adopted an inductive multiple case study approach to the research, which involves carefully analyzing real FLOSS developers' interactions for evidence of the coordination mechanisms in use. An inductive approach was indicated by our desire to extend theory for this new phenomenon. The rationale for the use of a case study approach was that case studies provide rich empirical data from a real setting, necessary for theory generation. In particular, Yin (1984) notes that case studies are particularly appropriate for answering *how* or *why* questions about current events in situations where the researcher has no control over the circumstances of the study.

Table 1. Projects Selected for Analysis.

	EGroupWare	**GAIM**	**Compiere**
Development Status	5 – Production/Stable	5 – Production/Stable	5 – Production/Stable
Programming language	PHP	C	Java
License	GNU General Public License (GPL)	GNU General Public License (GPL)	Mozilla Public License 1.1 (MPL)
Developer count	42	12	44

Sample Selection

Because the process of manually reading, rereading, coding, and recoding messages is extremely labor-intensive, we had to focus our attention on a small number of projects. We analyzed coordination mechanisms used among developers of three successful FLOSS projects, namely GAIM, eGroupWare, and Compiere ERP.

- GAIM is an instant messenger application that supports multiple platforms and protocols (**http://sourceforge.net/projects/GAIM/**).
- eGroupWare is a multiuser, web-based groupware suite with modules such as e-mail, address book, calendar, content management, forum, wiki, and so on (**http://sourceforge.net/projects/egroupware/**).
- Compiere is an ERP+CRM solution for small-medium enterprises covering areas such as order and customer/supplier management, supply chain, and accounting (**http://sourceforge.net/projects/compiere/** and **http://www.compiere.org/**).

The development status of the three projects is shown in Table 1.

The rationale for the selection of these three specific projects was that they all seemed to be successful at managing the contributions of multiple developers (the core developers noted in Table 1 plus many more peripheral contributors), thus providing relevant data for insight into coordination mechanisms for FLOSS development. We assessed success according to the criteria suggested by Crowston et al. (2003): these three projects are successful in that they have attracted numerous developers beyond the initial project founders, are continuing to release software, have numerous downloads, and have an active user community that provides feedback.

In addition, these projects differ in ways that allow for some interesting comparisons. GAIM is an end-user desktop application written in C, while eGroupware in a web-based server application written in PHP. As a result, we expect GAIM mostly to be used as is, but expect eGroupware to have a lower barrier to entry for developers and to be more often customized, potentially increasing problems in integrating these contributions. Compiere was originally a commercially developed product that later moved to open source development. Its history allows us to examine coordination problems that arise as a new set of developers begin working with an established code base and developer community. As well, one of the authors had extensive experience with proprietary ERP systems, providing a basis for comparison.

Data

The data used for the study were interactions on the main developer communication venue, either a developer mailing list or online forum. We chose these interactions because they are the communications between developers used to coordinate project development. To increase the comparability of our analyses, we examined messages from a similar period in the development life cycle for each project. We expected that coordination-related activities would occur more frequently around (and especially before) a major release, so we analyzed the period leading up to and immediately after the first open source release for each project.

- For GAIM, we selected messages posted to the "GAIM-devel" mailing list during August and September 2004 for analysis (GAIM 1.0.0 was released on September 16, 2004). The total number of messages was 710, posted by 85 individuals (11 were identified as developers according to the current developer list for GAIM, and 1 was identified as a former developer).

- For eGroupWare, we selected messages posted to the "egroupware-development" mailing list during October and November 2004 for analysis (version 1.0.00.006 of eGroupWare was released on November 18, 2004), which resulted in 665 messages total posted by 151 individuals (20 were identified as developers according to the current developer list).

- For Compiere, we selected messages posted to the Development Chat Forum from the January 1, 2001, to November 20, 2002, which covers the period up to and following the September 5, 2002, release of the Compiere Version 2.4.3a, the first version released after the move to SourceForge. Perhaps because this was a project transitioning from in-house development, the initial traffic on the developers' forum was sparse and we had to draw from a longer time period than with the other projects to gather the 315 messages we examined. The Compiere project had more defined roles: the total number of users posting was 57; project managers, 2; advisor/mentor /consultants, 3; developers, 6; and translators, 3.

All three projects selected are hosted on the SourceForge system (**http://sourceforge.net/**), making data about them easily accessible for analysis. Nevertheless, analysis of these data poses some ethical concerns that we had to address in gaining human subjects approval for our study. On the one hand, the interactions recorded are all public and developers have no expectations of privacy for their statements (indeed, the expectation is the opposite, that their comments will be widely broadcast). Consent is generally not required for studies of public behavior. On the other hand, the data were not collected for research purposes but rather to support the work of the teams. We have, therefore, followed the common practice of rendering all data anonymous by using pseudonyms in publications.

Analysis

For this project, we carried out a coordination-theory-based analysis of the developer e-mail interactions to identify the coordination mechanisms used in the process. We started by developing a coding scheme based on prior work by Crowston and Osborn (2003) on coordination theory. Crowston and Osborn describe three heuristics for identifying coordination mechanisms: matching activities performed against known coordination mechanisms, such as searching for duplicate tasks or task assignment; identifying possible dependencies between activities and resources and then searching for activities that manage them; and looking for problems that suggest unmanaged or incompletely managed dependencies. However, these approaches were developed for use in field settings and so the scheme had to be evolved for use in coding interactions in e-mail messages. In particular, we found that it was difficult to identify dependencies from the messages, so our attention focused instead on coordination problems and mechanisms. The coding system was evolved through discussion of the applicability of codes to particular examples, both in group meetings and as three coders worked on the same set of messages over the course of several months. The final coding system has 24 categories, organized into 3 high-level codes: task-task, task-resource, and resource-resource coordination mechanisms taken directly from the categories of coordination theory.[2] Through the process of iterating the coding system, it became apparent that the most interesting and novel aspect of the FLOSS development process as revealed in our sample was the approach to task assignment. Therefore, we focused our final efforts on this part of the coding system. All messages were double-coded for task assignment to allow computation of inter-rater reliability. The level of inter-rater agreement for coding the Compiere, eGroupWare, and GAIM projects were 0.893, 0.887, and 0.810 respectively, all above the usual rule-of-thumb acceptable level of 0.80.

Results: Dependencies and Coordination Mechanisms in Software Development

In this section, we discuss coordination mechanisms identified in the software development process. We then discuss differences between the FLOSS and proprietary processes as well as differences among the three FLOSS projects.

[2]Space does not allow complete description of the lower-level codes, but the coding system is available from the authors on request.

Common Output Dependencies Are Managed in a Similar Way

Many coordination problems and mechanisms appear to be broadly similar between proprietary and FLOSS development processes. For example, one common dependency is a *common output* dependency, meaning that two tasks create the same output. Avoiding such duplicate tasks is difficult if there are numerous workers who could be working on the same task, especially if they are distributed and have restricted opportunities to interact. For example, many users may report the same bug, but managers of a development group would likely prefer not to waste resources diagnosing and solving the same problem repeatedly. Crowston (1997) described how marketing engineers in a proprietary development organization would search a bug report database for each bug report to identify possible existing solutions or duplicate reports. *Looking for duplicate tasks* is a coordination mechanism for managing a dependency between two tasks that have duplicate outcomes. In the FLOSS process, users are encouraged to search the bug tracker database for duplicate problem reports before filing a new one. Nevertheless, there is no guarantee that users will search or that they will find a duplicate even if it exists, so duplicate bugs are reported and have to be identified and marked as such by developers. For example, an e-mail from GAIM developers pointed out that a problem report duplicated a known bug:

```
I suspect that this is the common "cygwin/bin in your path" issue.  If you
do have cygwin's bin directory in your path, it causes these symptoms as
GAIM tries to load cygwin's tcl84.dll for the tcl plugin loader.  You can
read more about it at http://GAIM.sf.net/win32 and in various threads on
the forum.
```

Another noted with regard to a patch:

```
Looks like this takes care of bugs #993985 and #1001031.
```

In FLOSS development, many developers might be working on the same part of the code, so this coordination mechanism can be applied more generally. For example, one developer on eGroupWare asked:

```
Is there someone who has any experience with including the app.  OWL
(owl.sourceforge.net) in eGW ?  I'm playing with the code right know, but
this wouldn't be necessary if someone did it before.
```

In a few cases, this kind of coordination was done ahead of time, as shown in these quotations:

```
Patches to update the svn tree can be submitted to me, please update this
thread with any work you are starting so we can avoid duplication.

I don't know how assignments work, but can I can dibs on this?  I don't
want to have any of your work duplicated, so I want to make sure that I
don't infringe on what someone is already working on.
```

In the Compiere project, many user questions come from the new users who are trying to install and use the software, so the project managers' and other senior developers' roles have evolved to include pointing people in the right direction to avoid duplication of efforts. In Compiere, we coded these kinds of references 29 times:

```
Yes ..  see doc http://www.compiere.org/support/install

There are also "hacker fixes" for the problem you mentioned.

Here are the links:   http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/
compiere/lib/postgresql.jar
http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/compiere/tools/l
ib/ocrs12.jar
```

Some Usability Dependencies Are Managed Similarly, and Some Differently

A second similarity is the management of flow dependencies, which arise when one task creates a resource that another task requires as an input. While in general a flow dependency implies the need for three different kinds of coordination mechanisms (flow, precedence, and usability), usability is often the most significant issue in the software development processes studied. Usability means that there must be some mechanism in place to ensure that the output of the first task is usable by the second task. An interesting usability constraint, common to both cases, arises from the flow dependency between the steps of finding a bug and fixing a bug. The output from the first step, bug reports from the general public, are often not detailed enough to be useful for developers performing the second step (Glance 2004). The low quality of bug reports discourages developers from relying on a bug reporting system, thus further decreasing the quality of the bugs. In response, some developers may take on the role of filtering or improving reports. In the proprietary process, marketing engineers check that problem reports are detailed enough to be used by the developers fixing the bugs (Crowston 1997). In the FLOSS process, developers might post follow-up messages to a report (either on the e-mail list or in the bug tracker database) requesting additional information. For example, one developer in eGroupWare asked a user to clarify a problem:

```
Does Outlook send some sort of error message?  What do you mean by crash--
does it stop responding, or close?  What version of Windows are you
running?  I'll try to figure this out asap.
```

Again, coordination mechanisms for ensuring usability can be applied generally. For example, a key problem in software development is requirements analysis, that is, ensuring that the software being developed meets the needs of the users. In most software development projects, requirements work is done by a dedicated team. In contrast, FLOSS development generally assumes that developers are users themselves and thus able to understand user requirements. Users may also read the developer lists and offer suggestions that way, as this quotation suggests:

```
I have read through this entire thread and seen a lot of suggestions for
how to implement this, and as an extensive _user_ of GAIM, let me say that
the best one that I have heard so far is the idea of tabbing the buddy
list interface.
```

In Compiere, we furthermore see that companies that use the software also use the company resources (time and people) to make the software fit their company or to enhance the software's functionality. Developers at a user company introduce the idea to the FLOSS team and coordinate efforts for that development:

```
Our company planned to work on a payroll and fixed asset module fof the
Compiere.  I wanted to know if anybody is interested in it?
```

The following conversation between an end user from a company and the project manger also shows how involvement of the companies as end users might change the coordination of the teams by bringing extra resources:

```
-  Hello, I saw on the page "product status" that serial numbers… are
   planned for the future.  Because we really need these functions, I
   would like to know when it will be implemented.

-  No plan yet.  Are you willing to sponsor it?

-  We are still in the evaluation period, but if we choose Compiere, why
   not…
```

Task Assignment Is Handled Differently

We next consider the coordination mechanisms that manage the assignment of tasks to actors. *Task assignment* is a coordination mechanism for managing the dependency between a task and an actor (a special case of a resource) by finding the appropriate actor to perform the task. There are numerous places where actors perform part of a task assignment process. For example, in the proprietary process described by Crowston (1997), customers give problem reports to the service center, which in turn assigns the problems to product engineers, who then assign them to software engineers. In addition, software engineers may assign reports or subtasks to each other. Interestingly, there appear to be significant differences in this area between proprietary and FLOSS development.

A key problem in task assignment is the basis for choosing the actor to whom to assign a task. For the proprietary process, the choice is often made by a project manager based on the specializations of the individual developers (Crowston 1997). Specialization allows developers to develop expertise in a few modules, which is particularly important when the engineers are also developing new versions of the system, and allows a single person to manage all changes to particular modules, thus minimizing the cost of integrating changes. However, the cost of such a system is the need for an elaborate process for assigning tasks to the appropriate developer.

In order to identify the key coordination mechanisms used for managing task assignment in FLOSS projects, we coded task assignment in terms of who was assigned the task and by whom. Five task assignment mechanisms are identified, as shown in Table 2. A striking feature of FLOSS development is how often developers introduce a task and offer to work on it, in effect assigning the task to themselves. For all three projects, self-assignment is the most frequent form of assignment. The following is an example of a poster self-assigning a task:

```
Thanx for the wonderfull help ...  Maybe an idea to make a 'hello world'
package for on the website ...  this kind of standard app ..  would be
verry Helpful for starting eGW developers (like me ;-)) If you'd like,
I'll make the package.
```

In a few cases, the offer to help is not connected to a particular task, but rather to claim responsibility for a general class of problems or just to announce availability:

```
Hi, developers, If you find any PHP5 related problems, please open a bug
report and assign it to me.

Well I just started 14 days of vecation in front of the telly and
computer.  So I guess I will do some coding these days.  =)
```

As well as volunteering, developers often propose tasks and ask for volunteers, explicitly or implicitly. For eGroupWare and Compiere, asking a certain person was the second most frequent mode of task assignment, followed by asking an unspecified person. For GAIM, asking an unspecified person is the second most frequent mode of task assignment, followed by asking a certain person. For example:

```
Can someone please do a brief test, replacing config.php with
newconfig.php?  If it works for a few people without causing problems, it
will help us in the long run.
```

Table 2. Frequency of Destinations of Task Assignment by Project

	Frequency		
Task Assignment Mechanisms	**EGW (%)**	**GAIM (%)**	**Compiere (%)**
Self assignment	37 (52.9)	60 (59.4)	16 (57.1)
Ask a certain person	15 (21.4)	18 (17.8)	9 (32.1)
Ask an unspecified person	12 (17.1)	22 (21.8)	1 (3.6)
Ask an outsider (not in the project development team)	0	1 (1.0)	0
Suggest consulting with others	6 (8.6)	0	2 (7.2)
Total of Task Assignments	70 (100)	101 (100)	28 (100)

Sometimes asking for volunteers is coupled with volunteering:

```
Our own project repository would require maintenance, bandwidth, and drive
space.  I've volunteered to do everything to get us started.  Volunteers
to help maintain would be appreciated.

I'm currently working through a port of my Bugzilla data into TTS, so feel
free to ask me for any more tips, suggestions, etc.  I'm happy to help, as
I'm going through the process myself!
```

Another interesting finding from the table is that sometimes team members are suggested to consult with others before they do a certain task, especially for eGroupWare. For example:

```
I only ask you to consult with the maintainer of the concerned app before
you commit something.  If you need to change something radical in the API
please talk it through Lars or me before (e.g., Mail it as proposal to the
developers list).

Sorry for that.  Have you talked to Bill about the patches?  I can't image
he's not willing to accept them.
```

Sometimes people don't directly ask others to do a task, but discuss who among several candidates can do it. For example:

```
Miguel our translation coordinator or I will get the translation via your
link and will commit them / include them in the distribution.  Ronald

Amir:   are  you  going  to  commit  the  translations  and  add  Thai  to
setup/lang/languages.php to enable Thai or should I take care of that?
```

Since self-assignment depends on the ability of an individual to contribute to the group, we investigated differences between developers and users in the type of assignment used, as shown in Table 3. The distinction between user and developer was assigned by comparing the poster of the message to the project's published list of developers.

Table 3. Comparison of Destination of Task Assignment by Developers and Users

	EGW		GAIM		Compiere	
	Developers (%)	**Users (%)**	**Developers (%)**	**Users (%)**	**Developers (%)**	**Users (%)**
Self assignment	25 (51.0)	12 (57.1)	28 (51.9)	32 (68.1)	8 (44.4)	8 (80.0)
Assign to a specified developer	1 (2.05)	2 (9.5)	3 (5.6)	5 (10.6)	3 (16.7)	1 (10.0)
Assign to a specified user	12 (24.5)	0 (0.0)	8 (14.8)	2 (4.3)	5 (27.7)	0
Assign to an unspecified person	7 (14.3)	5 (23.8)	15 (27.7)	7 (14.9)	0	1 (10.0)
Ask an outsider	0	0	0	1 (2.1)	0	0
Suggest consulting with other developer	3 (6.1)	2 (9.5)	0	0	0	0
Suggest consulting with another user	1 (2.05)	0	0	0	1 (5.6)	0
Suggest consulting with others	0	0	0	0	1 (5.6)	0
Total of Task Assignment Messages	49 (100)	21 (100)	54 (100)	47 (100)	18 (100)	10 (100)

The table shows that for all three teams, users rarely assign work to other users, but still often self-assign tasks. Especially for GAIM, the percentage of users is very high in self-assignment and some users offer to contribute several times. For example, one user in eGroupWare committed himself to helping implement features:

```
I am willing to help implement this feature, and have sometime :-) to give
away, so I started looking through the code.
```

In contrast to the separation of roles in the proprietary process, many users who post to the developer e-mail list prefer to solve the problem by themselves when they identify a task, instead of reporting it directly and just waiting for the responses. In other words, the formal division of users and developers does not necessarily constrain their behavior.

Discussion

Several points strike us as particularly interesting about these findings. A first observation is that FLOSS development shares a number of coordination mechanisms in common with proprietary development, such as those for managing common output dependencies or the usability of bug reports (alough there may be some differences in who performs these functions). This similarity is to be expected. Although the makeup of the teams and their approach differs, there is still significant commonality determined simply by the fact that both are undertaking software development.

The most striking difference is that community-based FLOSS development does in fact seem to rely less on explicit assignments of work, as has been suggested in the anecdotal literature on FLOSS development. First, we did not observe evidence of a hierarchy in assigning tasks in FLOSS. By lack of hierarchy, we mean that individuals do not command or direct others to work on a task as might a project manager. Instead, they use phrases such as “would you please,” “if you have time, can you…,” or even discuss how to assign a task instead of assigning it directly. Even assignment to a specific person is qualified: “If you’re interested you can …” or “feel free to figure out why and fix it if you like.”

Second, we observed broader participation in the work on tasks. Due to the openness of the FLOSS teams, active users can take on development tasks rather than the process being restricted to the official development team. Indeed, for all three teams, the most common form of task assignment was self-assignment, that is, volunteering to work on a particular task. These user-developers do not wait to be given something to do, but rather step forward to work on tasks that catch their interest. This form of self-assignment may be an emergent phenomenon. A participant in GAIM had this to say, “`I don't know how assignments work, but can I can dibs on this?`” In the absence of guidelines or rules, self-assignment may emerge to fill the need for a coordination mechanism to manage this part of the collaboration.

One likely explanation for these differences is that FLOSS development teams are substantially composed of volunteers. As a result, task assignment in FLOSS needs to be based primarily on personal interest, not specialization or ownership. Self-assignment appears to play a role in bolstering the legitimacy of the action suggested by the poster, making it more likely that the group will consent to the poster’s preferences. Legitimacy is the right of a participant to be heard and respected in a forum and a source of influence on the forum’s decisions. Formal groups with predefined memberships assess legitimacy up-front. In proprietary software development, the right to make suggestions for features is often reserved for the specification writers or customer-facing marketing engineers. On the other hand, for informal groups seeking new contributors, it is not possible, nor desirable, to make such prequalifications. The prevalence of self-assignment supports the anecdotal suggestion that in FLOSS projects in general, legitimacy is linked to action, the value that “code speaks louder than words.” After all, if someone is willing to use their own time to implement a feature or fix a bug (or even better, if they have actually already done so), it is more difficult for other members of the team to deny them that opportunity.

The FLOSS approach results in a task assignment process similar to the market approach suggested by Crowston (1997). In a market form of task assignment, a description of each task is sent to all available agents. Each evaluates the task and, if interested in working on it, submits a bid to work on the task and the task is then assigned to the best bidder. However, the FLOSS process differs in that there may not be an explicit assignment of the task; rather, a developer can choose autonomously whether to work on a bug. The use of this task assignment mechanism in the FLOSS development process is consistent with predictions of the effect of information and communications technologies (ICT). ICT makes it easier to gather information about available resources and to decide which resources to use for a particular task, thus favoring coordination mechanisms that are more information-intensive. At a macro level, Malone et al. (1987) suggest that decreased coordination costs favor more extensive use of markets, which usually have lower costs but require more coordination activities, over vertical integration, which makes the opposite trade-off.

Of course, reliance on this kind of assignment does have several drawbacks. First, anyone can choose to contribute to the project, even if they are not good at it or have no relevant experience. As a result, the quality of team member output often needs further investigation by developers and other users. In some projects, developers chose to forego these contributions rather than spend the time evaluating the output. Second, because multiple developers may be working on the same parts of the project, projects must develop code management practices that allow multiple changes to be integrated. Many projects rely on code management systems such as CVS, and the design of these tools has become a pressing topic in FLOSS discussions (see the recent discussion of source code control tools for Linux at **http://en.wikipedia.org/wiki/BitKeeper**).

Finally, proprietary task assignment and resource planning relies on the participants being reliable; the employment relationship, with its clear penalties for nonperformance, gives managers a reliable expectation that the work assigned will be carried out. By contrast, it has been suggested that FLOSS participants are relatively unreliable (Michlmayr 2004). The data in this study provide some support for this idea, although we did not directly measure the motivations or volunteer status of the participants so this cannot be fully confirmed. Much of the self-assignment is qualified with phrases like "if I have time," "if I get some free time," or, citing real-world time constraints, "I will submit this in a couple of day (have to finish some professional work first ... :/)." These caveats make sense in a volunteer activity where the service of individuals is on their own terms, but this uncertainty can create problems in the usability of output and timeliness of schedules. People finish their tasks according to time and interest, which may affect the effectiveness of the project. Because of uncertainty of code ownership, and the distributed nature of the participants, it is hard to track the status of developers' work, especially when it is self-assigned.

Conclusion

Software development involves many dependencies and a variety of mechanisms are used to manage them. For example, the possibility of duplicate tasks may be ignored or may be investigated before engineers attempt to solve the problem. Dependencies between tasks and the resources needed to perform them are managed by a variety of task assignment mechanisms, such as managerial decision-making based on expertise, workload in proprietary development organizations, or volunteering in FLOSS. The choice of coordination mechanisms to manage these dependencies results in a variety of possible organizational forms, some already known (such as change ownership) and some novel (such as voluntary selection of work by developers). Understanding the roles and tradeoffs of these mechanisms will help guide managers interested in transferring practices from FLOSS development into their own organizations and this paper provides the first empirical investigation of these practices in FLOSS development.

Our results suggest several avenues for future research. First, our results are based on just three projects. Studies of other projects are needed to confirm the pattern of coordination mechanisms reported here and to identify factors affecting the choice of mechanisms. For example, we expect that task assignment in company-sponsored projects works differently than in community-based projects. Further studies could examine the role of project culture, leadership, or power. As well, future studies should seek to identify FLOSS-specific coordination mechanisms used to manage other dependencies. For example, there has been some discussion of how FLOSS manages the usability dependency between system development and use that is usually managed by a formal requirements process (Scacchi 2002). Our study has only scratched the surface of this aspect.

Finally, future studies could examine the link between coordination and other group processes. For example, task assignment appears to play a role in the development of shared mental models for the project teams. "Assignment to unspecified person" provides a discursive and informal but continuous source of to-do items and the desired future of the project (e.g., "I'd really appreciate it if maybe someone could write the list suggesting the use of 'sound themes'" or "I think it'd be cool if someone modified the image"). Yamauchi et al. (2000) identified the practice of maintaining to-dos at various levels of specificity as a feature of FLOSS development. Similarly, assignment to a specific person also communicates the assigner's understanding of the other participants' skills and areas of knowledge. "Sarah, can you please clarify for me a few things about this design?" or "BTW, have you talked with Alvin at all? I think he has something like your global status drop-down somewhat implemented too" not only creates a task for Sarah or the question asker, but announces to the rest of the community that Sarah probably knows something about that design and that Alvin has experience with status drop-downs. In this way, practices that build shared mental models are embedded in coordination mechanisms and as such do not require explicit additional work for participants, minimizing the effort required to collaborate.

For managers of traditional software groups, perhaps the most interesting result is use of volunteering as an assignment mechanism. On their face, the concerns noted above about the problems with voluntary assignment suggest that the task assignment practices of FLOSS developers would be hard or undesirable to transfer to mainstream software development.

However, these coordination mechanisms may not be limited to volunteer projects only. Knoor-Cetina (1999) identified similar "gentle management" in the high energy physics (HEP) experiments at CERN[3]. She writes of "a marked self-organization of the experiments observed, a form of voluntarism" and relates a coordinator using a gentle approach in which he hopefully finds volunteers—somebody willing because they have a particular interest, something which happens whenever new tasks require attention (p. 179). She continues, "it is not 'morale' per se that is [the origin of self-organization or volunteerism], but the discourse that expresses necessities and interdependencies and allows for groups 'freely' (with a little nudging on the part of conveners and spokespersons) to respond to demands." Self-assignment and gentle assignment to others, then, are also found within mission-critical work involving the spending of substantial financial and scientific resources.

Another example of the use of these techniques within a nonvoluntary environment is the practice reported at Google where engineers are allowed one day a week to work on whatever they want to work on, as did 3M (see, for example, **http://www.oreillynet.com/pub/a/network/2005/03/16/etech_2.html**). The innovations in the Google Labs (labs.google.com) are said to be the result of these self-assigned activities. There is scope for the emergent practices identified in this study to be imaginatively applied in other contexts.

Acknowledgments

This research was partially supported by NSF Grants 03-41475 and 04-4468. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. This paper has benefitted from suggestions from Kathy Chudoba. The authors also thank the FLOSS developers who contributed to the research.

References

Abell, P. *The Syntax of Social Life: The Theory and Method of Comparative Narratives*, Clarendon Press, New York 1987.

Alho, K., and Sulonen. R. "Supporting Virtual Software Projects on the Web," in *Workshop on Coordinating Distributed Software Development Projects, 7th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, June 17-19, 1998, pp. 10-14 (available online from **http://ieeexplore.ieee.org/Xplore/dynhome.jsp**).

Britton, L. C., Wright, M., and Ball, D. F. "The Use of Co-ordination Theory to Improve Service Quality in Executive Search," *Service Industries Journal* (20:4), 2000, pp. 4: 85–102.

Crowston, K. "A Coordination Theory Approach to Organizational Process Design," *Organization Science* (8:2), 1997, pp. 157–175.

Crowston, K. "A Taxonomy of Organizational Dependencies and Coordination Mechanisms," in *Organizing Business Knowledge: The MIT Process Handbook*, T. W. Malone, K. Crowston, and G. Herman (Eds.), MIT Press, Cambridge, MA, 2003, pp. 85-108.

Crowston, K., Annabi, H., and Howison, J. "Defining Open Source Software Project Success," in *Proceedings of the 24th International Conference on Information Systems*, S. T. March, A. Massey, and J. I. DeGross (Eds.), Seattle, WA, 2003, pp. 327-340.

Crowston, K., and Kammerer, E. "Coordination and Collective Mind in Software Requirements Development," *IBM Systems Journal* (37:2), 1998, pp. 227-245.

Crowston, K., and Osborn, C. S. "A Coordination Theory Approach to Process Description and Redesign," in *Organizing Business Knowledge: The MIT Process Handbook*, T. W. Malone, K. Crowston, and G. Herman (Eds.), MIT Press, Cambridge, MA, 2003, pp. 335-370.

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, pp. 1268-1287.

Espinosa, J. A., Kraut, R. E., Lerch, J. F., Slaughter, S. A., Herbsleb, J. D., and Mockus, A. "Shared Mental Models and Coordination in Large-Scale, Distributed Software Development," in *Proceedings of the 22nd International Conference on Information Systems*, V. Storey, S. Sarkar, and J. I. DeGross (Eds.), New Orleans, LA, 2001, pp. 513-518.

[3]CERN is the European Organization for Nuclear Research, the world's largest particle physics center (see **http://www.cern.ch/**).

Espinosa, J. A., Lerch, F. J., and Kraut, R. E. "Explicit Versus Implicit Coordination Mechanisms and Task Dependencies: One Size Does Not Fit All," in *Team Cognition: Understanding the Factors that Drive Process and Performance*, E. Salas and S. M. Fiore (Eds.), APA, Washington, DC, 2004, pp. 107-129.

Faraj, S., and Sproull, L. "Coordinating Expertise in Software Development Teams," *Management Science* (46:12), 2000, pp. 1554-1568.

Glance, D. G. "Release Criteria for the Linux Kernel," *First Monday* (9:4), 2004.

Herbsleb, J. D., and Grinter, R. E. "Splitting the Organization and Integrating the Code: Conway's Law Revisited," in *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, May 16-22, 1999, 1998, pp. 85-95.

Herbsleb, J. D., Mockus, A., Finholt, T. A., and Grinter, R. E. "An Empirical Study of Global Software Development: Distance and Speed," in *Proceedings of the 23rd International Conference on Software Engineering* Toronto, Canada, May 12-19, 2001, pp. 81-90.

Knorr-Cetina, K. *Epistemic Communities*, Harvard Education Press, Cambridge, MA, 1999.

Kraut, R. E., Steinfield, C., Chan, A. P., Butler, B., and Hoag, A. "Coordination and Virtualization: The Role of Electronic Networks and Personal Relationships," *Organization Science* (10:6), 1999, pp. 722-740.

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

Malone, T. W., and Crowston, K. "The Interdisciplinary Study of Coordination," *Computing Surveys* (26:1), 1994, pp. 87-119.

Malone, T. W., Yates, J., and Benjamin, R. I. "Electronic Markets and Electronic Hierarchies," *Communications of the ACM* (30:6), 1987, pp. 484-497.

Michlmayr, M. "Managing Volunteer Activity in Free Software Projects," in *Proceedings of the 2004 USENIX Annual Technical Conference, FREENIX Track*, Boston, June 27-July 2, 2004, pp. 93-102.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. "Two Case Studies of Open Source Software Development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology* (11:3), 2002, pp. 309-346.

Ocker, R. J., and Fjermestad, J. "High Versus Low Performing Virtual Design Teams: A Preliminary Analysis of Communication," in *Proceedings of the 33rd Hawaii International Conference on System Sciences*, January 4-7, 2000, pp. 1019-1028 (available online from **http://ieeexplore.ieee.org/Xplore/dynhome.jsp**).

Raymond, E. "The Cathedral and the Bazaar," *First Monday* (3:3), 1998.

Scacchi, W. "Understanding the Requirements for Developing Open Source Software Systems," *IEE Proceedings Software* (149:1), 2002, pp. 24-39.

Seaman, C. B., and Basili, V. R. *Communication and Organization in Software Development: An Empirical Study*, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, 1997.

Wayner, P. *Free for All*,HarperCollins, New York, 2000.

Yamauchi, Y., Yokozawa, M., Shinohara, T., and Ishida, T. "Collaboration with Lean Media: How Open-Source Software Succeeds," in *Proceedings of the Conference on Computer-Supported Cooperative Work*, Philadelphia, PA, 2000, pp. 329-338.

Yin, R. K. *Case Study Research: Design and Methods*, Sage Publicatoins, Beverly Hills, CA, 1984.