Coordination processes in open source software development

The Linux case study

June 30, 2005 · Academic

Federico Iannacci


Abstract

Although open source projects have been subject to extensive study, their coordination processes are still poorly understood. Drawing on organization theory, this paper sets out to remedy this imbalance by showing that large-scale open source projects exhibit three main coordination mechanisms, namely standardization, loose coupling and partisan mutual adjustment. Implications in terms of electronically-mediated communications and networked interdependencies are discussed in the final sections where a new light is cast on the concept of structuring as a by-product of localized adjustments.

Introduction

The issue of coordination in open source software in general and Linux in particular has poorly been explored by scholars and practitioners alike. For instance, in a recent paper presented at the FLOSS (Free Libre Open Source Soft-ware) workshop, Crowston (2005) has maintained that “[i]n an OSS project, a range of coordination mechanisms seem to be used, but exactly what these practices are has not been studied in detail.” By the same token, Weber (2004: 11-12) has insightfully asked at the beginning of his book, “[h]ow and why do these individuals coordinate their contributions on a single focal point? The political economy of any production process depends on pulling together individual efforts in a way that they add up to a functioning product. Authority within a firm and the price mechanism across firms are standard means to efficiently coordinate specialized knowledge in a complex division of labor – but neither is operative in open source.”

Instead of studying the coordination processes characterizing all types of open source projects, this paper focuses on one of such projects, namely Linux, because coordination is remarkably more interesting and challenging in large-scale projects (Lanzara & Morner, 2003: 10) considering that size brings about a host of issues, such as the problem of information overload bearing on the project leader(s), the increased possibility of conflict, let alone the need to motivate a growing number of developers to keep momentum.

The remainder of this paper unfolds in the following fashion: the next section overviews the literature on open source software development by identifying two separate threads of thought; the following section (“The Linux case study”) highlights the methodology I have used to address the issue of coordination; “Coordination processes” spells out the coordination mechanisms characterizing the Linux kernel development, whilst the concluding section outlines the major contributions that might be derived from my argument.

Literature review

Although there is a burgeoning literature on open source software development, one can identify two main threads of thought, namely the social and the engineering stream (Feller & Fitzgerald, 2000). While the social stream analyzes the social dynamics of open source software development by drawing on such ideas as gift economy (Raymond, 1999), social structure (Healy & Schussman, 2003), user-based innovation (vonHippel, 2001), complexity/chaos theories (Kuwabara, 2000; Nakakoji, et al., 2002) and transaction costs (Benkler, 2002), the engineering approach attempts to situate the open source model within the context of other systems development methodologies (Feller & Fitzgerald, 2000).

Despite this enormous research effort, it seems that the issue of coordination has not yet been fully addressed to the point that scholars contend that “little is known about how people in these communities coordinate software development across different settings, or about what software processes, work practices, and organizational contexts are necessary to their success” (Crowston & Howison, 2005).

But what is coordination? Broadly defined, coordination consists of protocols, tasks and decision mechanisms designed to achieve concerted actions between and among interdependent units be they organizations, departments or individual actors (Thompson, 1967; Van de Ven, et al., 1976; Kumar & van Dissel, 1996). Although there is a promising theory that concentrates on interdependencies between activities (Malone & Crowston, 1994), in the reminder of this paper I set out to investigate interdependencies between and among actors (Thompson, 1967; Weick, 1979) rather than activities because open source projects feature relatively-low task interdependencies due to their modularity. More specifically, the question I attempt to
address is the following: how do open source developers organize their interdependencies despite being distributed across space and time? However, as it was outlined in the previous section, coordination in large-scale open source projects is remarkably more interesting and challenging than small-scale ones (Lanzara & Morner, 2003: 10). By taking the Linux operating system as a case in point, this paper attempts to spell out the way developers manage their interdependencies in large-scale open source projects[2].

The Linux case study

Linux is a Unix-like operating system started by Linus Torvalds in 1991 as a private research project. In the early history of the project Torvalds wrote most of the code himself. After a few months of work he managed to create a reasonably useful and stable version of the program and, therefore, decided to post it on a Usenet Newsgroup to get a great number of people to contribute to the project[3]. Between 1991 and 1994 the project size developed to the point that in 1994 Linux was officially released as version 1.0. It is now available free to anyone who wants it and is constantly being revised and improved in parallel by an increasing number of volunteers (Kollock, 1999).

Like many other open source projects (Moody, 2001 ; Weber, 2004), Linux exhibits feature freezes from time to time whereby its leader announces that only bug fixes (i.e., corrective changes) will be accepted in order to enhance the debugging process and obtain a stable release version. The Linux kernel development process, therefore, may be decomposed into a sequence of feature freeze cycles each signaling the impending release of a stable version.

Given my concern with coordination processes, I set out to use a longitudinal case study (Pettigrew, 1990) as my research design because longitudinal case studies enable researchers to collect data which are processual (i.e., focused on action, as well as structure over time), pluralist (i.e., describing and analyzing the actors’ multiple worldviews), historical (i.e., taking into account the emergence of interaction patterns over time) and contextual (i.e., examining wider cultural, institutional and organizational practices that partake in the construction of contexts).

Since February 2002 represents a point of rupture in the lifespan of the Linux development process due to the official adoption of BitKeeper (BK), a proprietary version control tool, by Torvalds, this paper endeavors to analyze the events surrounding the October 2002 feature freeze, the first freeze exhibiting the parallel adoption of two versioning tools, namely BK and CVS (the Concurrent Versions System), on the assumption that this freeze stands for a test bed for future freezes[4].

In analyzing such events, I looked at communication threads concerning such themes as the patch submission procedure, the bug reporting format, the workflow for incorporating new patches in Torvalds’s forthcoming releases, etc. I also studied who interacts with whom in order to spot broader interaction patterns. I decomposed each thread into sets of two contingent responses between two or more perceived others, thus taking Weick’s (1979) ‘double interact’ as my unit of analysis.

To sum up, I took communication cycles of double interacts as my unit of analysis because each double interact refers to an act, a response and a subsequent adjustment between two or more individuals so that each communication cycle can be closed. Although one case study cannot be readily generalized to the universe of open source projects, in dissecting the Linux kernel development interaction patterns I was after analytical generalizations rather than statistical generalizations (Yin, 2003).

Coordination processes

If coordination can be conceived of as the management of interdependencies between and among organized actors, how do the Linux kernel developers go about organizing their work interdependencies? My longitudinal case study suggests that there are three coordination processes developers resort to, namely standardization, loose coupling and partisan mutual adjustment.

Standardization encompasses uniform and homogeneous procedures developers enact for problem definition or problem solving (Kallinikos, 2004). Such procedures, in turn, refer to the predefined bug reporting format and patch submission routine that developers are to follow whenever they are posting new software misbehaviors (i.e., bugs) or new features on the mailing lists[5]. The Linux documentation archive, for instance, paints a standardized patch submission procedure which expressly mandates that the Linux kernel developers:
1. create patches in DIFF format, that is to say in a way that extracts the difference between a modified set of files and an original set of files. The modified version is normally reproduced with a manual procedure called PATCH provided that the original version and the difference are at hand (Yamaguchi, _et al._, 2000);

2. thoroughly describe their changes;

3. separate their logical changes into their own patches so that third parties can easily review them;

4. carbon copy the the Linux Kernel Mailing List (LKML);

5. submit their patches as plain text or, if they exceed 40 Kb in size, provide the URL where they stored their patches;

6. provide the name of the version to which their patches apply and include the word PATCH in their subject line[6].

Apart from standardization, loose coupling stands for another critical coordination mechanism that allows developers to manage the workflow for incorporating new patches into Torvalds’s forthcoming releases. While standardization pools the developers efforts together and makes them available across spatio-temporal contexts[7], loose coupling coordinates various social subsystems through weak ties.

To exemplify the latter point, consider the following comment posted on the LKML by Linus Torvalds, the chief maintainer, in response to Rob Landley’s suggestion to create a “Patch Penguin” to filter incoming software features to Torvalds himself:

“Some thinking, for one thing. One ‘Patch Penguin’ scales no better than I do. In fact, I will claim that most of them scale a whole lot worse. The fact is, we’ve had ‘Patch Penguins’ pretty much forever, and they are called subsystem maintainers. They maintain their own subsystem, i.e., people like David Miller (networking), Kai Germschewski (ISDN), Greg KH (USB), Ben Collins (firewire), Al Viro (VFS), Andrew Morton (ext3), Ingo Molnar (scheduler), Jeff Garzik (network drivers) etc etc… A word of warning: good maintainers are hard to find. Getting more of them helps, but at some point it can actually be more useful to help the ‘existing’ ones. I’ve got about ten-twenty people I really trust, and quite frankly, the way people work is hard-coded in our DNA. Nobody ‘really trusts’ hundreds of people. The way to make these things scale out more is to increase the network of trust not by trying to push it on me, but by making it more of a star-topology around me. In short: don’t try to come up with a ‘Patch Penguin’. Instead try to help existing maintainers, or maybe help grow new ones. THAT is the way to scalability.”[8]

Although Landley was pushing for an artificial design intervention aimed at institutionalizing a new figure in the Linux development process, namely the ‘Patch Penguin’, Torvalds’s remarks eloquently stress that the only way to make the network scale is to work with a limited number of people who, in turn, “work with their own limited number of people”[9]. The upshot of this process is a different way of organizing activities that resembles a heterarchy rather than a hierarchy where the workflow is “kind of a star [shape], with Linus in the center, surrounded by a ring of lieutenants, and these lieutenants surrounded by a ring of flunkies, may be the flunkies surrounded by a ring of flunkies’ flunkies, but the flow of the information is through this star, and there are filters. So that you end up with Linus getting stuff that most of the time he doesn’t have to work on very hard, because somebody he trusts has already filtered it” (Moody, 2001: 179). This workflow is illustrated below:

---

**Legend:**

T=Torvalds;

TL=Trusted Lieutenant;

D/U=Developer/Uploader;
What, then, are heterarchies? Heterarchies are nested hierarchies (Jen, 2002) where the concept of hierarchy does not mean official channels or chains of command from the top down; “[i]nstead, in this context, hierarchy means only that subsystems can differentiate into further subsystems and that a transitive relation of containment within containment emerges” (Luhmann, 1995: 19)[10]. Put differently, heterarchies can be conceptualized as loosely-coupled systems considering that, since each system is a part of the whole, as well as being a whole in its own right (Mitleton-Kelly, 2003), one can envisage an ensemble of systems where the coupling between systems is weak or loose because the interactions between systems are less direct and less frequent than those within systems (Orton & Weick, 1990; Beekun & Glick, 2001)[11].

Another way to grasp this idea is to think of loosely-coupled systems as less richly-connected networks where perturbations are slow to spread and/or weak while spreading (Weick, 1976) considering that they exhibit weak ties between their subsystems and strong ties within them (Simon, 1969: 183-216; Luhmann, 1995)[12] as it is shown in Figure 2 below:

But what is the implication of this logic? Linux exhibits the following traits typical of loosely-coupled systems (Cf. Weick, 1976: 5):

---

**Fig. 2: Heterarchies as loosely-coupled systems**
1. Slack times, that is times where there is an excessive amount of resources relative to demands. Many open source scholars have argued that modules and parallel development lead to duplicate efforts (Feller & Fitzgerald, 2002; Gläser, 2003; Lanzara & Morner, 2003);

2. Occasions when any one of several means will produce the same end Again, many scholars have already pointed to this phenomenon. Lanzara and Morner (2003), for instance, use the qualifier “trash” as a colorful expression to stress that many efforts simply duplicate existing features, thus literally being a waste of time and precious computing skills[13];

3. Weakly connected networks in which influence is slow to spread and/or is weak while spreading The BK example is a case in point where influence is slow to spread considering that key trusted lieutenants have not yet implemented this tool;

4. Relative lack of coordination, slow coordination or even coordination that is being reduced as it moves through the system. If there is a relative lack of coordination, slow coordination or even reduced coordination through the system how are loosely-coupled systems organized? My point is that it is the strength of their weak ties (Granovetter, 1973) that glues these systems together;

5. Relative absence of regulations. At present there are very few, if any, regulations that administer this fledgling phenomenon although it seems that the European Union (EU) is trying to enact new regulatory prescriptions on open source development[14];

6. Planned unresponsiveness. Linus Torvalds, for instance, expects constant re-submission of patches on behalf of his trusted lieutenants. Thus, it is as though the system has been planned to be unresponsive[15];

7. Causal independence. The Linux development process exhibits a reversed relationship of causality between means (i.e., actions) and ends (i.e., intentions) because goals are established retro-spectively rather than prospectively[16];

8. Poor observational capabilities on behalf of viewers. Tightly-coupled ‘elements’ are much easier to observe because they respond to if/then laws of cause and effect which do not seem to apply to the Linux case study and, more broadly, to loosely-coupled systems;

9. Infrequent inspection of activities within the system. In the Linux kernel development process activities are hardly inspected because there is no formal authority;

10. Decentralization. Activities are highly decentralized because people can follow their interests and needs;

11. Delegation of discretion. People are free to pursue their own interests;

12. Absence of linkages. I interpret this trait to stand for a minimal number of relationships between subsystems (Burt, 1992; Kogut, 2000), and;

13. The observation that an organization’s structure is not co-terminus with its activity. The workflow for incorporating new patches has triggered a loosely-coupled social structure which is not the by-product of localized activities but the outcome of size and complexity[17].

Despite the trade-off between exploration and exploitation (Weick, 1979; March, 1991; Levinthal, 1997), loosely-coupled systems are viable alternatives to such a trade-off because they are able to search the space of possibilities through localized adaptations while maintaining local stabilities which ignore limited perturbations elsewhere in the system (Glassman, 1973).

Partisan mutual adjustment and structuring

If the Linux kernel developers resort to coordination by standardization and loose coupling to organize their pooled and sequential interdependencies respectively, how do they go about coordinating their networked interdependencies? Compared to team interdependence where work is fed back and forth within a team of individuals acting jointly and simultaneously (Van de Ven, et al., 1976), networked interdependence refers to a potentially-unbounded set of developers who work disjointly and asynchronously due to the information and communication technologies (ICTs) in use, as well as other full-time commitments.

I submit that networked interdependencies are coordinated in accordance with the principle of emergence insofar as coordination is the by-product of ordinary decisions undertaken in the pursuit of local interests (Lindblom, 1965; Warglien & Masuch, 1995). Put differently, whenever developers are disjointly and asynchronously feeding their work back and forth among
Consider, for instance, the following table outlining the interaction patterns concerning a specific module of the Linux kernel development process. Table 1 refers to threads (i.e., communication cycles) that were started on the LKML during the period spanning from October 2002 to December 2002. I have discarded those threads having a single message and/or involving just one developer because I was interested in the frequency of interactions between at least two developers. To illustrate, if one takes the column dated 11/11 entitled “kexec for version 2.5.47,” one should find out a total number of messages equal to 26 where Biederman, the Informal Kexec Maintainer, made 12 postings, Almseberger 3 postings, Piffer 2 postings and the perceived others 9 postings, 2 of which were made by Bhattacharya, a developer whose involvement with the kexec system call has grown over time[18]. What do these data tell us?

First, Biederman operates as the Informal Maintainer of the kexec system call because he has started the project and keeps updating it as new kernel versions become available (for instance, kexec for 2.5.42, kexec for 2.5.44, etc.). This point is not trivial: in this project there is an individual who is fully committed because of a public and yet irreversible choice (Weick, 1993)[19]. His decision, in other words, is the expression of a free choice of developing or maintaining a patch set which is repeatedly posted on a mailing list for the noticing of others. Hence, there is a choice (i.e., a decision) which is public and irreversible because this patch set is repeatedly updated on the LKML by its Informal Maintainer. This logic, in turn, implies that the Linux kernel developers can take up maintenance roles within the community that feature three distinguishing traits, namely openness (i.e., they are free to develop whatever patch set is of their interest), irreversibility (i.e., once they have started working on a certain patch set, they cannot go back because it is in their interest to keep it in sync with the kernel whether their patch is in the main kernel or not) and visibility (i.e., the other kernel developers know that these people operate as the Informal Maintainers of certain patch sets). Not only can Informal Maintainers start their own patch sets; they can even make variants of Torvalds’s releases (Axelrod & Cohen, 1999) which co-evolve in a “friendly rivalry”[20] and spur long-term adaptability through better search of the space of possibilities (Stark, 2001; Iannacci, 2003)[21].

Second, commitment creates a social network around the Informal Maintainer which is open to contributions from everybody. Yet despite being open, the network features quite strong ties between few developers because of their frequent interactions. Moreover, the kexec social network features strong ties among Biederman, Piffer and Almseberger. These relationships of networked interdependence can somehow be outlined in the following fashion:

![Fig. 3: Kexec system call interactions[26]](image-url)

<table>
<thead>
<tr>
<th>Perceived others (Bhattacharya)</th>
<th>5</th>
<th>6</th>
<th>3</th>
<th>1</th>
<th>9</th>
<th>8</th>
<th>2</th>
<th>4</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Piffer</td>
<td>5</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Almseberger</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Biederman</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>12</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

**TIME**

- 12/30
- 12/30
- 12/30
- 12/30
- 12/30
- 12/30
- 12/30
- 12/30
- 12/30
- 12/30
- 12/30
- 12/30

Legend:

B = Biederman (Informal Maintainer)
po = Piffer
a = Almseberger

Fig. 3: Kexec system call interactions[26]
Third, and finally, far from being a constant, the strength of these ties is a variable because the frequency of interactions is bound to change over time as the developers' interests evolve. For instance, despite having interacted quite frequently with Pfiffer and Biederman in the time span under investigation, Almesberger's interests or workload changed over time to the point that his last posting on the LKML dates back to April 3rd, 2003 without involving the kexec system call[22]. Bhattacharya, on the contrary, was quite disenchanted by the kexec system call at first, but became gradually more involved with the project in the year 2003 to the point of turning into a stable point of reference for the Informal Maintainer:

> “...Suparna [Bhattacharya] this should be a good base to build the kexec on panic code upon. Until I see it a little more in action this is as much as I can do to help. And if this week goes on schedule I can do an Itanium port...” Eric Biederman[23]

This logic, in turn, implies that the by-product of the Linux kernel developers' interactions can be described as a social structure made up of bundles of dyadic ties (Barley, 1990) which are bound to evolve as the frequency of the interaction changes. Thus, the kexec social structure evolved over time into a new social network which can be represented in the following fashion:

![Fig. 4: Kexec social network](image)

Legend:
- B = Biederman (Informal Maintainer)
- po = Perceived others
- b = Bhattacharya
- p = Pfiffer
- Dashed lines = weak ties
- Solid lines = strong ties

![Fig. 5: Kexec new social network](image)

This last point is further corroborated by the fact that Torvalds's network of trusted lieutenants itself can be considered an emergent social structure where the strength of the interpersonal ties is indeed a variable rather than a constant:

> “You will never figure that out [i.e., Torvalds’s network of trust], it isn’t predefined. It reshapes itself on the fly, and is really defined by what is going on at any given time. That said, it’s usually possible to figure out how [who] the main maintainers are, and what to send where, just don’t hope to ever nail that down in a rigid structure. It’s not rigid...”[24]

I submit that in highly-unstructured situations where work is fed back and forth among a potentially-unbounded set of loosely-coupled developers, coordination pivots around these minimal social structures because they stabilize the communication processes by making such interactions more orderly, more predictable and more organized.

In other words, should one consider postings occurring in 2003, it is very likely that one will find out that Pfiffer is constantly testing the kexec system call patches that Biederman develops over time.

Notice additionally that Thompson’s (1967) hypothesis whereby higher degrees of interdependence call for the additive use of all coordination processes being investigated seems to hold up quite well in the Linux case study with the exception that there is a large decrease in the use of loose coupling, and much more reliance on ad hoc structuring as a by-product of partisan mutual adjustment. This is because the former mechanism requires a large number of developers to emerge considering that it is a feature of organized complexity (Kallinikos, 1998)[25].
Conclusions

This paper aims at dispelling the issue of coordination in large-scale open source projects. There are three main contributions that can be drawn from the aforementioned analysis. First, although the electronic means of communication are context-reliant media (Sproull & Kiesler, 1991; McKenny, et al., 1992), my analysis shows that, to some extent, the Linux kernel developers are able to reproduce the missing context by resorting to such standardized procedures as the patch submission and the bug reporting routine. Is this a startling finding? The media literature has so far maintained that electronic contexts enhance the problem of equivocality, thus requiring face-to-face communications as complementary media (Daft & Lengel, 1986; Nohria & Eccles, 1992). My reading of the Linux case study questions this argument by showing that what is needed is a structural substrate whereupon social interaction can occur. This, in turn, means that the real issue is a problem of grade or minimal threshold: once developers resort to uniform and homogeneous procedures for problem definition and problem solving, they can reproduce the missing context to a certain degree. At this point, rich communication media are no longer required because developers can interact across spatio-temporal boundaries despite the fact that the electronic media in use filter out the social context cues.

Second, although Van de Ven, et al. (1976) have already felt the need to extend Thompson’s (1967) concept of reciprocal interdependence by introducing the notion of team interdependence, my reading of the Linux case study shows that their influential opinion needs further extension: while team interdependence refers to workflows where “the work is acted upon jointly and simultaneously by unit personnel at the same point in time” (Van de Ven, et al., 1976: 325), I submit that a new concept needs to be entertained by organization scholars, namely the idea of networked interdependence featuring a potentially-unbounded team where work is performed in a disjoint and asynchronous fashion in between various full-time task assignments.

Third, and directly related to the previous point, networked interdependence is coordinated through a structuring process which is the by-product of partisan mutual adjustments rather than a separate and centralized set of coordinating decisions (Lindblom, 1965). To reiterate, while pursuing their localized interests, developers coalesce around few projects that are appealing to them. By so doing they inadvertently come to interact with a select number of programmers with whom they are bound to create relatively stable ties on the basis of decisions which are public and relatively irreversible because of the high frequency of interactions. I submit that these emergent structures operate as pivotal coordination mechanisms because they make social interactions more orderly, more predictable and better organized over time. Therefore, my claim is that “genuine parallelism” (Warglien & Masuch, 1995: 4) is the distinguishing feature of the Linux kernel development process where truly-independent parallel search (Cohen, 1981) by multiple developers walking many directions at once (Iannacci, 2003) is corroborated by the emergence of relatively small social clusters which attend to specific areas of the source code on the basis of relatively stable social ties.

To put a final seal to this paper, if organizational form includes both structure and process (Crowston, 2005), I submit that open source projects can be considered as new forms of organizing whose coordination processes are the by-product of partisan mutual adjustments while their structures may be conceived of as heterarchies or loosely-coupled systems, provided that there is a large number of developers involved, as well as a modular architecture and a feedback loop channel be it a mailing list or a newsgroup.

References


