MAKING AND SHARING KNOWLEDGE AT ELECTRONIC CROSSROADS: THE EVOLUTIONARY ECOLOGY OF OPEN SOURCE

Giovan Francesco Lanzara[a]
Michèle Morner[b]

[a]Dipartimento di Organizzazione e Sistema Politico,
University of Bologna, Italy
lanzara@spbo.unibo.it

[b]School of Business Administration,
Catholic University of Eichstätt-Ingolstadt, Germany
michele.morner@ku-eichstaett.de

Session J-3

Abstract

Based on the analysis of developer mailing lists of two large-scale open source projects, we argue that, in open source development, processes of knowledge making and sharing exploit the structuring properties of high density, massive interaction for evolutionary purposes. The mailing lists reveal patterns of activity and resource distribution that exhibit ecological features. A high number of agents meet and exchange knowledge at 'electronic crossroads' within a complex web of software artefacts and communication tools that play a critical role in supporting such knowledge ecology. Electronic artefacts foster variety, mediate human interaction, and replace formal organizational mechanisms. Our findings show that the evolutionary features of knowledge making and sharing in virtual environments challenge current ways of conceptualizing knowledge processes within and across organizations

Keywords: knowledge making and sharing, ecology, evolution, interaction, inscription, open-source software projects.

Making and sharing knowledge at electronic crossroads: the evolutionary ecology of open source

Giovan Francesco Lanzara *
Michèle Morner **

* Dipartimento di Organizzazione e Sistema Politico
Università di Bologna, Italy
lanzara@spbo.unibo.it

** Wirtschaftswissenschaftliche Fakultaet Ingolstadt
Katholische Universitaet Eichstaett, Germany
michele.morner@ku-eichstaett.de

Abstract

Based on the analysis of developer mailing lists of two large-scale open source projects, we argue that, in open source development, processes of knowledge making and sharing exploit the structuring properties of high density, massive interaction for evolutionary purposes. The mailing lists reveal patterns of activity and resource distribution that exhibit ecological features. A high number of agents meet and exchange knowledge at 'electronic crossroads' within a complex web of software artefacts and communication tools that play a critical role in supporting such knowledge ecology. Electronic artefacts foster variety, mediate human interaction, and replace formal organizational mechanisms. Our findings show that the evolutionary features of knowledge making and sharing in virtual environments challenge current ways of conceptualizing knowledge processes within and across organizations

Keywords: knowledge making and sharing; ecology; evolution; interaction, inscription; open-source software projects.

Suggested track: J (Knowledge and Information Technology)

1 Introduction

In open-source software projects as many as thousands of skilled programmers and users collectively develop code online *via* the Internet in a decentralized, highly interactive, apparently unmanaged process (Raymond, 2001; Lerner and Tirole, 2001; Kogut and Metiu, 2001). In spite of their spatial dispersion, fluid participation and loosely coupled organization open-source software projects are surprisingly effective in tapping and coordinating the talents of many spatially dispersed contributors, delivering competitive, high quality software (Raymond, 2001; von Hippel, 2001; Kogut and Metiu, 2001; Metiu and Kogut, 2001). Distributed knowledge resources of the most diverse kinds are gathered from multiple sources, productively used to develop software, and swiftly circulated across geographical and organizational boundaries to feed a large variety of innovative products (Kogut and Metiu, 2001; von Krogh et al., 2003; von Hippel, 2001).

As a model of knowledge making and sharing, open-source software development elicits a number of questions that pose a theoretical challenge to current ways of conceptualizing knowledge processes within and across organizations (Kogut and Zander, 1992; Tsoukas, 1996; Eisenhardt and Santos, 2000; Patriotta, 2003):

- How can knowledge making and sharing take place in such an extremely decentralized form of project organization without the usual governance structures and without permanent membership in a classical sense, but nevertheless result in the creation of a joint product?
- What are the basic mechanisms underlying coordination and the creation of knowledge in open-source software projects, and where are they embedded?
- What is the role of artefacts in the process of knowledge making and sharing?

This paper is an attempt to give an answer to the above questions. Our research approach moves away from current perspectives in a two-fold way. First, we regard open-source software projects primarily as 'interactive systems' rather than organizations (Goffman, 1983; Luhmann, 1995), nevertheless displaying a number of organizational characteristics (Morner, 2003). In open-source software projects, processes of knowledge making and sharing exploit the structuring properties of high density, large-scale interaction for evolutionary purposes. Knowledge is the emergent outcome of multiple-agent interaction and is based on communication (Luhmann, 1995). Second, we argue that in order to grasp how knowledge-related processes happen in open-source software projects we should look at the technology rather than

at the organization. Interactive systems are deeply intertwined with a web of electronic artefacts and communication tools that support programming, development and knowledge-based activities at large. Technology embodies many of the organizational rules and means for governance in open-source software projects, hence becoming a critical pathway to the understanding of collective task accomplishment, coordination and knowledge making.

Our argument is based on an in-depth analysis of the development activity carried out within two large-scale open-source software projects: the LINUX kernel and the APACHE HTTP Server development projects. To explore knowledge dynamics we analyse what open-source developers concretely do in their everyday mundane practices: writing programs and having e-mail conversations about programming. We focus on the aggregate outcome of individual behaviours and on the role artefacts play in supporting knowledge making and sharing. Artefacts are seen not as static ‘repositories' of packaged knowledge but rather as dynamic vehicles of evolving knowledge. Although in open-source software projects a multiplicity of artefacts exists, here we only focus on a specific electronic artefact, the developer mailing lists of LINUX and APACHE. Such lists reflect the ongoing development activity carried out by programmers. We have analysed the structure and dynamics of discussion threads as they are imprinted in the lists. The threads reflect both design knowledge about the making of the software and knowledge about coordination and distributed organizing.

Our findings lead to a view of knowledge making and sharing practices as a complex ecology of multiple heterogeneous elements and interactions, where the evolutionary mechanisms of variation, selection, and stabilization keep the system in a dynamic balance (Baum and Singh, 1994; Aldrich, 1999). Particularly, we stress the crucial role of the mechanism for generating variety in knowledge systems. The paper is organized as follows: in the next section we articulate our theoretical framework. In section 3 we briefly describe our data sources and research method. Then the analysis of the electronic mailing lists is developed in section 4, where the body of data is presented. Next, based on our interpretation of the data, we reconstruct the dynamic pattern of open-source software projects as it emerges from the findings and develop the idea of knowledge making and sharing as an ecological process.

2 Building a theoretical framework: interaction, inscription, and evolution

2.1 Open-source software projects as interactive systems

If for once we put aside an organization-theoretic bias, open-source software projects, at their most basic level, can be usefully regarded as interactive systems (Goffman, 1983; Luhmann, 1995). According to Luhmann (1995: 412-416), interaction refers to communication between those present, and interactive systems emerge when several individuals engage in related communication and perceive the fact that they are engaged in communication. When presence ends, the system ends. However, in open-source software projects participants are not 'present' in a classical (physical) sense, only virtually. They are present in the form of perceiving each other in an electronic medium – the Internet. In principle, they have the possibility to follow each and every communication because of its documentation in mailing lists, bulletin boards or newsgroups web sites. In other words, they become 'co-present' when they connect and 'talk' throughout the web from their workstations, otherwise they disappear.

Although not 'organized' in the classic sense, when they reach a critical mass interactive systems may reveal interesting structuring properties. First, they show an attentive core, which consists, for example, in a common subject or task that must be attended to. The subject provides the agents with a focus of interest and the system with a structure, as the boundaries of the subject regulate participants' contributions (Luhmann 1975: 24), and who does not attend to or participate in its development cannot influence it. Participation and influence on the subject depend on the amount of attention and time that the agent is able or willing to put on the subject. This functional selectivity, integrating participants' ability for attention and remembering, turns time to structure, that is, the limited ability to participate essentially creates the structure of the interactive system. Thus, some foci of attention emerge and disappear through time. Clusters of activity coalesce and disband, and the agents' convergence to a focus is alternatively enacted and discontinued in the interaction space.

In open-source software projects common subjects or themes are quite typical of communication and development processes. They attract and 'anchor' the attention and skills of the programmers. Attention, being a scarce resource, is intrinsically selective. Agents cannot pay attention to an unlimited number of themes simultaneously, but must have priorities that produce sequences. As a consequence, communication threads are generated. Threads act as focal points (Schelling, 1960). They have ordering properties in the interaction space and may persist over time:

individuals that would otherwise interact randomly arrange and coordinate their interactions around the focal point. Also, threads are characterized by a next-next-next pattern: a message calls for the next around a specific programming task or discussion theme and along an ongoing stream of activity and sense. The close sequencing of communications generates a self-sustaining process fostering inter-temporal connectivity. By directing or diverting attention participants create temporal sequences of entries and exits around a specific thread. Threads may emerge unexpectedly and suddenly disappear, but when they reach some stability and persistence over time they play an important part in modularly structuring the project. As we shall see in section 4, the thread pattern stimulates reciprocity and becomes the basis for coordination and knowledge making in open-source software development.

Yet, open-source software projects also exhibit characteristics that are more typical of formal organizations. First, together with fluid participation there is also stable membership, usually limited to a core of professional developers who are in control of critical functionalities or development areas of the project. Second, together with largely unmanaged processes, there are governance mechanisms at work. If for some respects open-source software projects have been regarded as chaotic systems (Kuwabara, 2000), nevertheless most projects rely upon simple decision making rules, both for programming and communication. Authority is allocated to make critical decisions in restricted areas. In addition to that, there are rules governing transactions both within a single project and between the project and its institutional environment. Third, differently from pure interactive systems, open-source software projects are able to build up memory: they keep track of their products and development processes through online, quasi-automatic documentation. Such feature allows for inter-temporal travelling throughout the process, thus helping project continuity, identity and sense making (Weick, 1995). Through such processes of continuous tracking and re-tracking, memory becomes a powerful mechanism for stabilizing and reproducing the activity system. Fourth and last, similar to organizations open-source software projects are equipped with representation mechanisms which allow them to directly communicate with their environment (and be recognized by it), both for symbolic and commercial purposes. If on the one hand these organizational features make open-source software projects interesting hybrids, on the other hand they do not really have a dominant or pervasive role in the control of distributed actvities, and taken alone would not be strong enough to account for the impressive performance of large-scale projects both as task-oriented production systems and as knowledge making and sharing mechanisms. Most of the organizational gear is invisible in open-source software

projects, being draped in the technical gear. In open-source software projects conspicuous components of ‘organization’ are embodied in software artefacts. Therefore, if one wants to search for organization, s/he’d better look at the technology.

2.2 Artefacts as inscriptions of knowledge and agency

The 'organization' of open-source software projects is embedded in the technological gear to such an extent, that it is impossible to account for basic knowledge making and sharing processes without focusing on the technology. Using a literary analogy, a number of authors see technology as an *inscription* of human agency and knowledge (Akrich and Latour, 1992; Latour and Wolgar, 1979; Latour, 1992; Joerges and Czarniawska, 1998; Patriotta and Lanzara, 2003). By *inscription* is meant the act (or sequence of acts) by which humans cast relevant components of their agency and knowledge into artefacts to which action programs and capabilities are delegated (Latour, 1992). As a result of *delegation*, artefacts become holders and dynamic vehicles of human agency, therefore replacing humans in doing things and performing functions in complex networks of human and non-human agents (Akrich and Latour, 1992). Introducing the concept of inscription seems to be an appropriate step in a context where the agents’ main work activity consists in *writing lines of code* that make up software programs and in *writing e-mail messages* that are posted on the web. In open-source software projects, different kinds of artefacts inscribe different kinds of knowledge and agency. For example, technology is an inscription in two different ways: as a dense web of multi-various software objects and as an electronic media for programming and communication.

A first instance of inscription is represented by software artefacts. Interaction among agents in open-source development projects occurs primarily in a network-mediated computer environment populated with an array of electronic objects and tools that Scacchi calls 'software informalisms' (Scacchi, 2001). These web-based artefacts inscribe different kinds of knowledge and help create a large-scale environment for programming and information sharing (Iannacci, 2002: 13). In many cases they fulfil multiple functions, being at the same time the products of development work, the tools of the trade by which development work is carried over, and even the media through which interaction and communication among the agents can take place. In open-source software projects knowledge work is extensively mediated by artefacts, which play a crucial role in distributed knowing and organizing. In our research we have examined the developer mailing lists as an electronic communication artefact inscribing software-based protocols and procedures that allow specific interactions while inhibiting others,

make possible specific ways of developing software jointly, and enact specific modes of knowledge making and sharing. Software artefacts and tools populating open-source environments are loosely integrated or recombined within and across the ongoing practices and processes. Rather than being a fully coherent set of tools, each having its own specific functional destination, they resemble a loosely connected collection of available objects that happen to be there in a permanent state of flux, being continuously assembled and discarded.

A second mode of inscription lies in the information infrastructure. The peculiar features of the open-source phenomenon would be difficult to grasp without paying attention to the information technology infrastructure supporting human agency and interaction. Open-source software projects live in an electronic media that confers specific properties to interactive systems and to the ongoing development practices. First, as the literature on computer-mediated communication has pointed out (Eveland and Bikson, 1988; Kling and Scacchi, 1982; Kollock, 1999), the media allows for asynchronous communication, agents' ubiquity, extended network-based transactions. These features are constitutive of the social practice (and the social order) of open-source. The interactive systems of open-source software projects are largely inscribed in (and supported by) the Internet, the basic information infrastructure. The Internet enhances social connectivity and facilitates its conversion to task-oriented collective action in a cost-effective way, allowing fast communication feedback and dramatically accelerating production and testing cycles. By the same token the Internet allows for easy storage and fast travelling and transfer of all sorts of electronic artefacts. It is a place where programmers and users can find all kinds of work tools and relevant information. Agents are 'wired' to such information infrastructure and their programming and communication practices are molded by it. Interaction, communication and agency become Internet-based and Internet-specific, to the point of becoming unthinkable independently of it. The Internet becomes a generalized medium for knowledge-related work. To its communication protocols and procedures an enormous amount of coordination is delegated, which otherwise would have to be provided by explicit governance mechanisms. Consequently, due to the Internet, creation and distribution of knowledge need very little mediation by market or corporate forms of governance, but directly exploit the connectivity properties of the society. In open-source software environments no knowledge making and sharing would be possible without such electronic gear and no theory of knowledge making could ever be developed without incorporating the Internet as a constitutive base of it, as a '*Gestell'* (Ciborra and Hanseth, 1999).

2.3 Evolutionary mechanisms: variation, selection, stabilization

Large open-source software projects, when they reach a critical mass, are less the product of deliberate management and planning and more the evolutionary outcome of a complex interplay of different processes and activities. Open-source software projects become powerful instances of how variation can be effectively combined with selection and stabilization for knowledge making purposes. Variation is the engine of the process of software development and innovation, keeping the process open to novelties and opportunities (Neff and Stark, 2002). It mainly comes through human agency and is at the core of basic learning processes in organizations and social systems (March, 1991; Aldrich, 1999). Variation enables software products and development processes to learn from the environment. It is an unbalancing mechanism that tends to push the system off its path. Knowledge creation would hardly be possible without variations. For example, in a knowledge-intensive process such as software development better or high quality software is software that has successfully encoded multiple sources of variation. Indeed, in open-source software projects the pressure of variation is so high that a legitimate question can be raised: How can the system handle so much variation? How can stability be granted to products and processes? Selective and stabilizing mechanisms are needed to balance the production of variety.

Selection reduces variety by eliminating uninteresting or inconsequential variations and focusing only on the ones that should be kept. Selection always works *ex post,* after the effects of variation have been experienced and tested (Aldrich, 1999). While selection makes variety manageable, at the same time it creates redundancy by producing unexploited opportunities, arrangements and solutions. These are not definitely discarded but set aside to be eventually re-entered into the development process for further re-combinations and re-uses at later times. Selection splits what is kept from what is left out. Its outcome depends upon value priorities and criteria of relevance, and this is what makes it a knowledge making mechanism in a development process where multiple moves and solutions are always available. A wide array of selective mechanisms is at work in open-source software projects. They can be explicit, such as authority-based decision rules and voting procedures, or else implicit, such as limited attention resources and technology-embedded filtering devices.

Finally, stabilization allows for the retention, accumulation and reproduction of successful experiences. In order to stabilize a selected outcome some repetition and codification are needed (Barley and Tolbert, 1997). Stabilizing mechanisms are built into the communication technology itself. They create memory, standards, rules,

behavioural patterns, structures and meanings. The emergence of stable system components facilitates and speeds up system evolution (Simon, 1969).

3 Data sources and research methodology

Case selection. Our data come from two projects: the LINUX kernel project and the APACHE HTTP server project. These projects were selected for two reasons: first, they are in a mature and stable phase of development. Second, a huge number of participants are involved in both projects, what makes issues of knowledge creation and coordination more interesting than in smaller projects. Focusing on only two cases has the advantage of increasing the depth of the analysis. As we can see from the data, the amount of activity and participation at LINUX is comparatively higher than at APACHE. However, in spite of the different goals, size and contents, the similarities in the two projects are more conspicuous than the differences.

Data collection. In order to tap contextual project information and build an overall picture of the development work we first analysed the *projects' web pages*. We examined the projects' history, structure, aims, core activities, version control tools, and the legal framework in the form of license arrangements. As a second step, we conducted twelve *personal interviews* in two rounds with four LINUX developers and two APACHE developers, each lasting from one to two hours. In part, the interviews helped us to check and eventually correct the picture that we were developing through our navigations across the web. But our main sources of data gathering were the *e-mail conversations* archived in the projects' mailing lists. The lists report the ongoing conversations and transactions among developers working on the open-source code. Although projects have also other mailing lists for bug reports, user support and announcements of code changes, the general discussion of development topics (for example technical details, errors, project design, emerging architecture, and announcements) takes place at the developer lists.

Data analysis. The activities recorded in the mailing lists are tracked and analysed both qualitatively and quantitatively. The units of the e-mail analysis are communication sequences of the developers in the form of e-mail threads. We define a thread as a sequence of e-mail messages around a common discussion theme with a distinct heading, for example a question or the posting of a new patch. Each first mail of a thread is expected to generate conversational activity that potentially extends into the following days, weeks, sometimes even months. Agents produce the thread by asking questions, giving answers or generating other communication activities via e-mail.

Using open coding (Strauss and Corbin, 1990), we developed a coding scheme for the content of the first e-mail of a thread. This provided eleven different first mail message types, including such items as reporting errors of the software ('bugs'), providing new parts of the software ('patches'), asking questions concerning the use of the software, and announcing new versions or new patches. We created a database of all threads including names of the thread initiators, date and time for posting the different e-mails, and content of the first mails at LINUX for the period November, 15th – November, 30th, 2002, and at APACHE for the period May 15th – May 30th. The chosen period for each project roughly corresponds to the project's yearly average activity (number of mails). In selecting the periods we assumed that, as the mailing lists reflect mundane everyday conversational and crafting activity of the programmers, what happens in a relatively short time period in the project's history can be taken as a plausible example of the general pattern of behaviour and knowledge making. In other words, we assumed that the pattern of ongoing interactions would tell us something about the dynamics of knowledge making and sharing in the projects. In order to test this assumption we also performed the data analysis for one single 'ordinary' day (November 30th for both projects) and found a similar pattern. The complete database includes 4600 single e-mail contacts in 1256 threads for LINUX initiated by 573 participants, in APACHE 594 e-mails in 143 threads initiated by 65 participants (see table 1).

Table 1. Basic data for LINUX and APACHE

	Number of Threads	**Number of Mails**	**Number of (initiative) Participants**
LINUX	1256	4600	573
APACHE	143	594	65

4 Threads of conversations: Sharing knowledge at electronic crossroads

In their daily practice developers entertain conversations, which are recorded in the mailing lists. Outcomes and problems of the programming work are discussed and refined in multiple ongoing streams of e-mail conversations[1]. Mailing lists are virtual work environments where various kinds of transactions among multiple agents take

[1] Conversations evoke an oral activity, but indeed these e-mail conversations, although they keep a seemingly oral form, are written down and edited within an electronic medium: such feature makes them a peculiar form of communication.

place (Lee and Cole, 2000). Developers and users use the mailing lists and other software-based communication devices to exchange ideas and documents, discuss problems, post new patches and solutions, make requests for help, launch announcements about new software features and products, etc. As knowledge objects mailing lists are manifold. First, they are virtual construction sites where individual programmers, linking from their remote workstations, jointly conduct their development and problem solving work (that is, they are places where people engage in the practice of jointly designing the software)[2]. Second, they are meeting places where information is found and exchanged, problems and solutions are discussed, and agreements are made (that is, places where people talk about the work they do, inquiry into it, and negotiate what to do next). Third, they are web logs where the history of open-source software projects is recorded (that is, places where the development work and the talking about it are documented).

Thus, an analysis of these artefacts is critical to understand the knowledge dynamics of an open-source software project, because it reveals *both* the time structure of the conversations that an undefined set of programmers entertain around specific coding issues *and* the knowledge content of the conversations. At the same time the mailing lists give us information about the structure of social interaction supported by the communication technology. These artefacts are dynamic, ever changing repositories and carriers of problem solving knowledge within and across projects and sub-projects. Through them, knowledge is circulated all over the projects and the programming environment. This is why we like to think of them as 'crossroads' where people meet, exchange information, and attend to common tasks. Basically, in the threads we find the footprints of the developers' moves and their aggregate behaviour over time. In discussing the data below we highlight the interplay between variation, selection, and stabilization in the process of knowledge making and sharing.

In both projects, although in different degrees, we notice a high production of variety, due to free accessibility, multiple agents, high number of started threads including a lot of mails with new problems and solutions (see again table 1). Fundamentally, the developer mailing lists are *accessible to anybody* who wants to register and post his comments. Day after day, *multiple agents* exchange messages and generate activity around multiple issues concerning the software. Whatever the message, it will be

[2] The developer mailing lists do not tell anything explicit about the actual making of the software in which programmers engage at their computer desk. This is largely inaccessible by an external observer, although it is perhaps partly visible in the structure of the source code. Nevertheless the conversations and communications refer continuously to the programmers' practice.

posted on the list or board and circulated across the network. By entering the conversations around specific issues users and developers generate a thread. Threads are given life and structure by time, reciprocity, and attention. The *number of threads* generated in the analyzed projects builds up the basis of variety. In the chosen period, at APACHE 143 threads with altogether 594 e-mails were initiated by 65 participants. At LINUX even more 'mass' is generated: 1256 threads including altogether 4600 mails were initiated by 573 participants. At LINUX 78,5 threads on average are created per day with a range of activity from 46 to 125, while in APACHE the average is 9,5 threads per day with a range from 2 to 22 threads. Not all intervening agents have experience, but the high variety guarantees the presence of a critical mass of experienced people. Such variety constitutes a large, available reservoir of knowledge resources that can feed the development process at any time.

Concerning the content of the communication, variation is produced by *new solutions* on the one hand and *new problems* on the other hand. We coded the first e-mail of a thread according to its content (see section 3 and figure 1). *Problems* are questions and problems in using the software (Q), the identification of errors – called 'bugs' (B), and general questions concerning further modules, versions, etc. (GQ). *Solutions* are new patches that are provided for further development of the software (P), suggestions for improvement of certain aspects (SI), and comments or answers to former threads (C). The data show that most of the communication in the developer mailing lists concerns new patches (P) that are provided for further development of the software (39% for LINUX, 38% for APACHE) or questions about problems using and developing the software (26% for LINUX, 27% for APACHE). It is interesting to notice that at both projects more threads (53% at APACHE and 48% at LINUX) begin with solutions offered to different problems (SI, C, P) than with new problems (Q, B, GQ) (at both projects 35%).

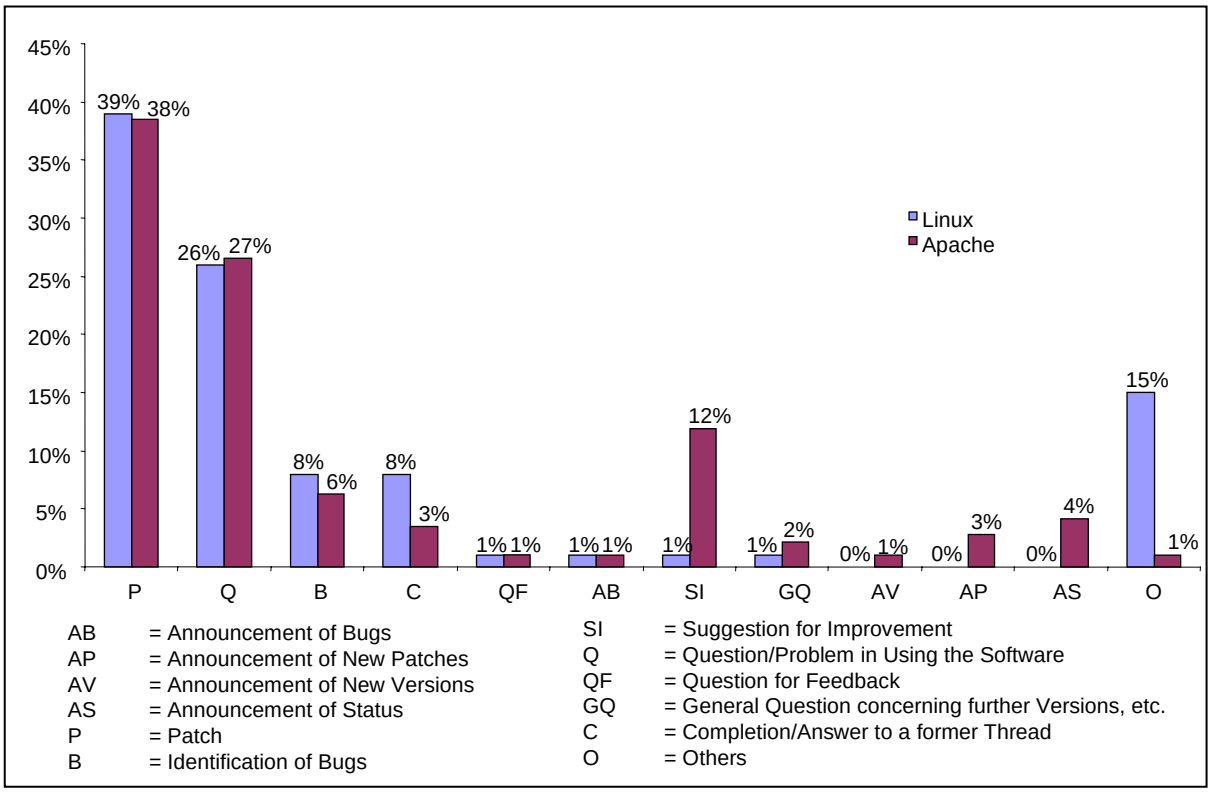

Fig. 1. Content of first mails per thread for LINUX and APACHE

Variety entering the process is counteracted by effective selection mechanisms. A large amount of information comes in, but only a little finds its way through the system and gets processed. Selection produces a remarkably skewed pattern of activity. Threads emerge selectively from generalized interaction, signalling some purposeful activity. Most of them are intermittent, ephemeral, volatile objects. Just minimal coalescent structures, fed by interaction (Weick 1993). In the following diagrams and table we show the effects of selection on the distribution of threads with respect to participation, activity and duration. The average figures show few participants per thread, few mails per thread, and short lifetime of threads. Even if the overall interaction is potentially all-to-all and every developer or user enjoys free access to all information on the web, the actual *participation* is very selective. Very few threads have more than a handful of participants. In APACHE we found that only 18% of the analysed threads have more than three participants, at LINUX only 14% (see figure 2).

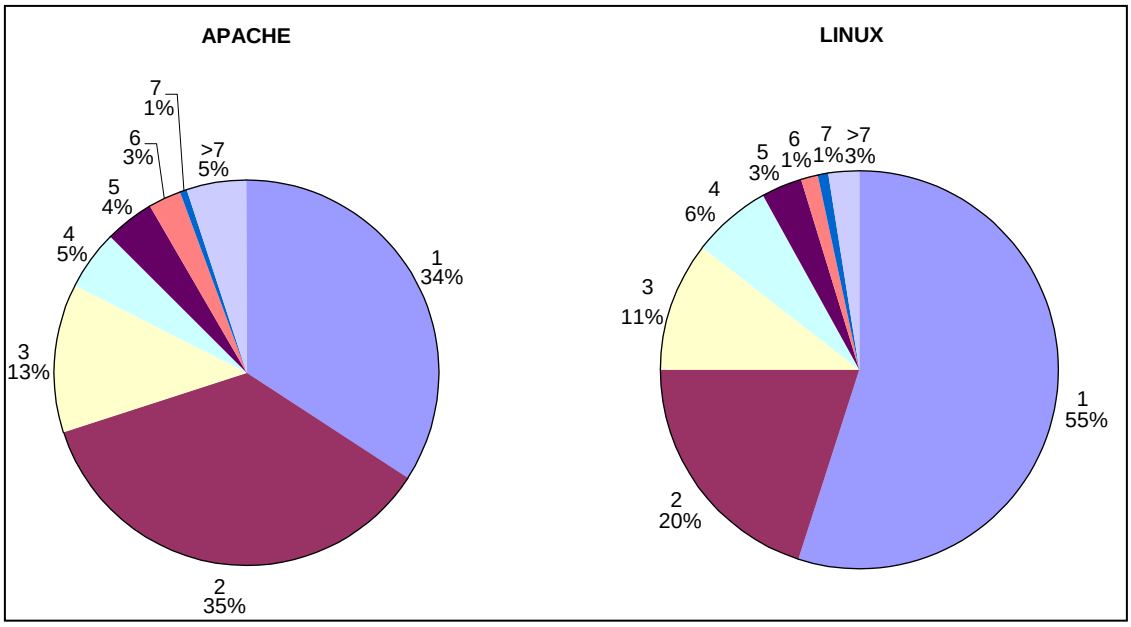

Fig. 2. Participants per thread (in %) for LINUX and APACHE

The average number of participants per thread is very low (2,2 for LINUX and 2,6 for APACHE). The average number of *mails per thread* is low, too (3,7 for LINUX and 4,2 for APACHE), with a minimum of one for both projects (messages without answers), but with remarkably high maxima at LINUX (see table 2).

Table 2. Averages for LINUX (Nov., 15th-30th, 2002) and APACHE (May, 15th-30th, 2002)

	Average number of participants per thread	**Average number of mails per thread**	**Average number of days per thread**
LINUX	2,2 (max= 36; min=1)	3,7 (max= 95; min=1)	2,9 (max=135; min=1)
APACHE	2,6 (max=16; min=1)	4,2 (max=38; min=1)	1,4 (max=17; min=1)

At LINUX as many as 75% of the analysed threads have less than four mails per thread and only 9% of threads have more than eight mails (see figure 3). In APACHE the pattern is the same, although the figures are a bit different: 71% of the threads have less than four e-mails per thread and only 12% reach more than 8 mails per thread. If we track the number of mails per thread, we notice that, especially at LINUX, a high percentage of first mails are left unanswered and do not have a follow up (LINUX 47%; APACHE 33%). In these cases, the thread is made of one participant and one mail, so basically there is no thread at all. Considering the content of the first mail, at LINUX even 39% of direct questions are left unanswered (24% at APACHE). The death rate of threads is impressive, but this does not necessarily mean a similar death rate of agents. They can always return and initiate new threads that eventually will be continued.

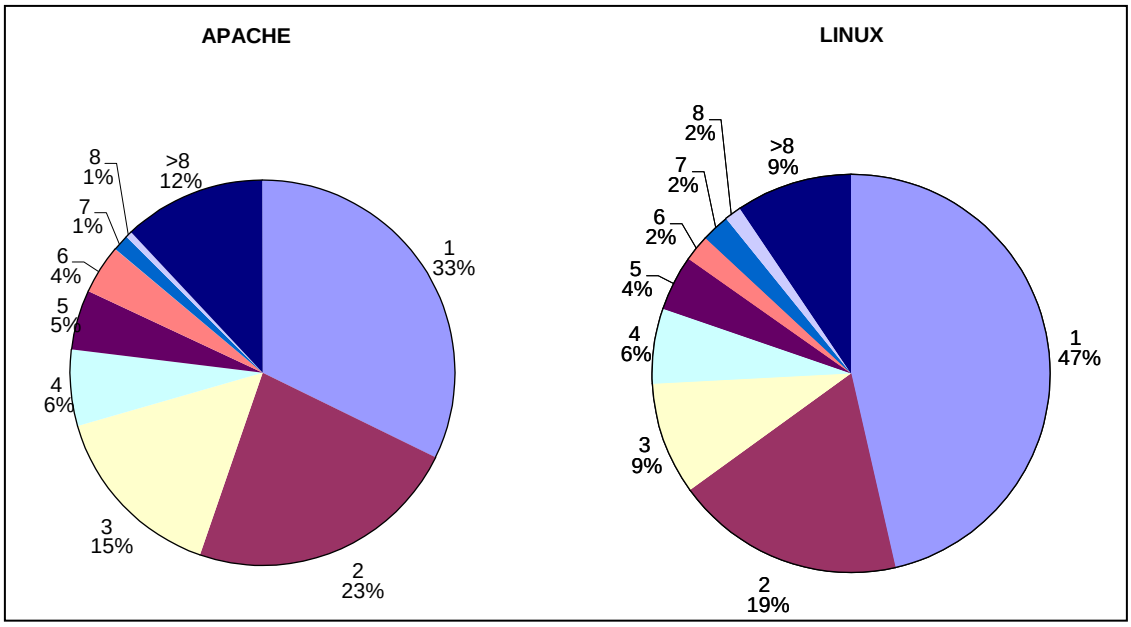

Fig. 3. Number of mails per thread for LINUX and APACHE

The average *lifetime* of a thread is generally very short: 2,9 days in LINUX, 1,4 days in APACHE, with a minimum of one day for both projects and maxima of 17 for APACHE and over one hundred for LINUX (see table 2 again). Most of e-mail conversations never really come to form a stable and lasting thread. In LINUX only 30% and in APACHE 19% of threads reach an age of more than one day (see figure 4). Taken individually threads tend to be short-lived, ephemeral structures, but as a whole they make persistent bundles or streams of varying thickness or thinness.

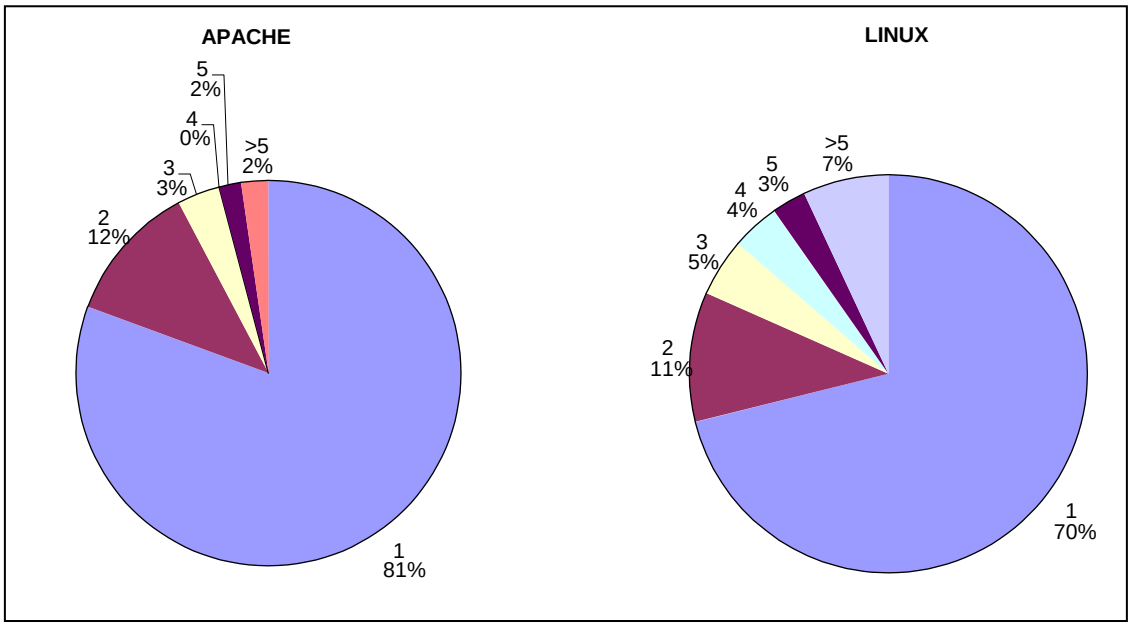

Fig. 4. Days per thread (in %) for LINUX and APACHE

Selection means as well dissipation. As the data show, there is a great amount of dissipation in the activity system. Basically, a lot of development effort is 'eaten up' throughout the process. It becomes trash. The reasons for this can be many: limited attention resources, perceived irrelevance, implicit filters inscribed in the technology. Dissipation in open-source software projects is not necessarily a source of inefficiency, because it produces redundancy that can be usefully exploited in the overall system dynamics: a distributed reserve of low cost knowledge and attention resources is available to feed project development needs at any time. Specific patches capture the attention of a few and discussion starts. Activity clusters around a few issues or themes at a time, on which attention is selectively focused, generating clustering of agents. Someone is always ready to pick up the job. Of course, there is a real possibility that an important signal, warning, question, or original solution is neglected, or that people pursue tracks that lead nowhere, but the high mass of participants providing simultaneous scrutiny minimizes the chance, overall. In this connection, it is interesting to highlight the different figures for variation and selection in LINUX and APACHE. The very same mechanisms are at work in both project, but in APACHE there seems to be slightly less dissipation than in LINUX. On average in APACHE more participants discuss with more intensity in shorter periods of time. This leads us to think that in APACHE connectivity is tighter and communication is more stable in time, or even that the process is more efficient. This is due perhaps to the different nature of the task, or to smaller number of participants, or to higher professionalism of the core developers. Conversely, the higher participation and dissipation rates in LINUX activity could well not be an indicator of the instability and inefficiency of the project but of LINUX broader popularity and easier accessibility.

Only in a limited number of cases the communication is *stabilized* and coalesces into conspicuous and relatively persistent threads. Few selected threads live for long. When that happens, they signal some more organized and purposeful work around a persistent theme or modular component. The longest threads are 17 days at APACHE, and 135 days at LINUX (see table 2 again). In the mailing lists we found mainly three aspects stabilizing the communication: recurrent participation of same participants, documentation and nested conversation. Usually, in longer threads the very same agents *participate recurrently*. For example at LINUX the threads with five mails are written by 3,4 participants on average. Threads with 19 mails are written by only eight developers on average. At APACHE three agents participate on average to threads with five mails. Also, the communication via mailing lists is stabilized by its quasi-automatic *documentation*. Documents and edited conversations archived on the

websites provide stability and continuity to otherwise tenuous and volatile relationships. The mails are archived and everybody can have access to them, eventually retrieve them and refer to them at any point of the development process. Knowledge making and sharing is definitely facilitated by the availability of ubiquitous documentary artefacts that mediate people's interactions and support their work. In other words, stability and continuity are built into the technology and in the computer-based communication and documentation protocols. One special way to use the intrinsic documentary quality of edited communication via e-mails is to enact *nested conversations*. Typically, parts of a previous message to which the following message refers are inserted into the new one. Iannacci (2002) labels such interactions 'dialogic negotiations' which imply a continuity and interdependence among messages as indicated by the inclusion of previous messages or pieces of messages, thus creating 'metaphorical conversations'. In our view the nesting of conversations are a visible instance of how potentially relevant bits of knowledge can circulate within and across projects and be shared by a variety of agents to the purpose of making knowledge.

Finally, to complete our analysis of threads we need to say something about the dynamic features of the process. Figure 5 illustrates the overall stream of threads at APACHE in the two-week period studied. The threads are calculated day by day and are symbolized by the arrows, whereas the small crosses signal single e-mail messages with no follow up.

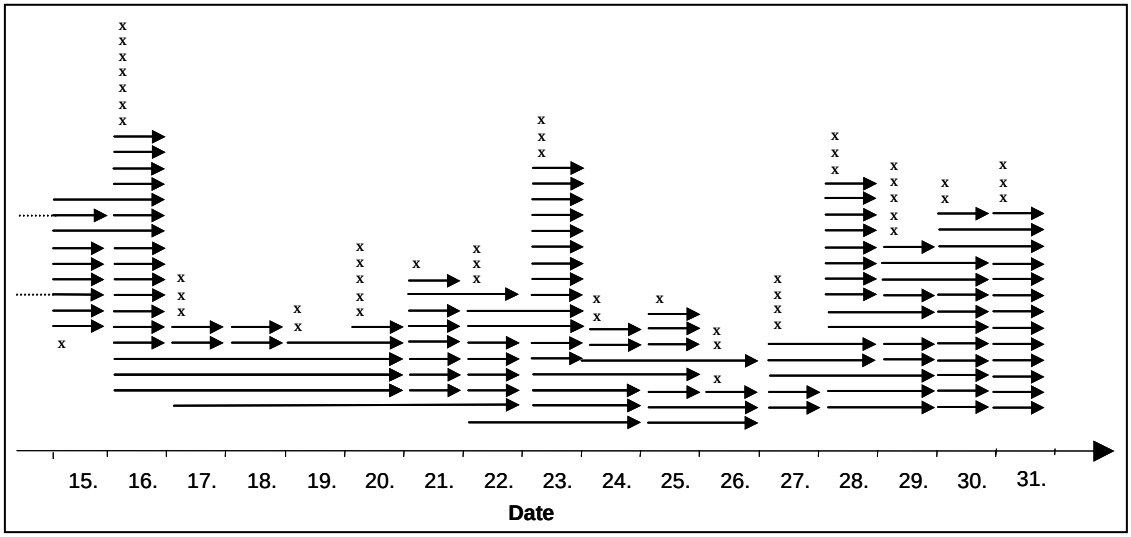

Fig. 5. Ongoing emergence and disappearance of threads for APACHE (May, 15th-30th, 2002)

The figure shows the ongoing emergence and disappearance of threads over time. As we have already said, threads are ephemeral entities. Most of them live the life of a

mayfly. Robust and durable threads emerge very rarely but the flow of communication is never completely discontinued. The single threads go off and on, but the overall fabric never collapses. At any point in time in the development of the project there is always a bundle of active threads that signal and carry on activity at varying degrees of intensity. Agents can go off the stream, but can always re-enter later on. Thus the inter-temporal continuity and stability of the project are assured by a flow of communication along a main stream of sense- and knowledge-making.

5 The emerging pattern: toward an ecology of knowledge

What pattern do the data point at? What do they tell us about knowledge processes in open-source software projects? How can these findings enrich our view of organizing and knowledge making? We have analysed mailing lists as critical artefacts that make large components of the programmers' design practice visible to observers within an electronic media. In the mailing lists we can find, posted on a website, traces of the distributed activities of a large number of geographically dispersed programmers engaged in a common development task. Choosing to focus on the mailing lists as an empirical object we wanted to track knowledge work related to the making of the software at the aggregate level, as it is imprinted in the structure of the agents' interactions.

5.1 Reconstructing the pattern: evolutionary features

At the aggregate level our data show a distinctive and consistent pattern of behaviour. The specific features characterizing the development process lead us to think of knowledge processes taking place in large-scale open-source software projects as being evolutionary and ecological. Artefacts such as mailing lists play a prominent role as dynamic vehicles of knowledge. In this section we draw a synthetic picture of the overall phenomenon, as it emerges from the analysis performed in the previous section.

First of all, a large-scale open-source software project, such as LINUX or APACHE, relies upon an endogenous mechanism for generating variety. Such mechanism is inscribed in the interactive system, supported by the technology, and institutionalized by the copy-left arrangement that allows open access. The project feeds on a constant source of variety produced by the participants' e-mail messages, actions and entries into the system. The agents bring in new problems, solutions, perspectives, and new software objects, eventually hiding bugs. The enormous amount of incoming variations,

in the form of new threads and mails, compared to what actually gets processed throughout the system, generates novelty and multiplicity, constituting the exploration side of the development and knowledge making process (March 1991). Variation explores multiple knowledge resources. Of course, not all agents have the same talent and skill but, because of the high numbers, a critical mass of experienced people is always available at any point in time.

The figures and tables show the pressure of variety, but they also show how variety is harnessed by selective mechanisms that admit and process only a few variations out of the many. The outcome of the process is a remarkably skewed distribution of threads, participants, activities, and durations of threads. The distribution points to the classic Zipf's power/rank law that characterizes population ecologies (Zipf, 1949). Out of multiple communications only a few have a follow up and coalesce in long lasting threads signalling stable and cumulative activity. Most of the threads are soon discontinued and potential resources are re-addressed to further threads or turned into trash. At a first glance, this would signal inefficiency, sloppiness, and a high level of dissipation. However, the economic argument doesn't stand up to deeper scrutiny. Dissipation means free resources that are unexploited *now* but can be re-entered and re-used *later* in the process. The participants' intentions and actions intersect with one another and are deflected at the local level, generating disorder and unstructuredness that call for increased attention and further action. Thus new forms and stable patterns can be created out of a chanceous event. The project doesn't have a clear and stable direction, it just 'rolls on', branching in multiple directions at once.

Trash means redundancy. The high degree of parallelism in the process tends to generate multiple solutions and patches that compete with one another: one is chosen; the others are discarded or put aside. But they may run together for a long time before being dropped. For each discussion thread or communication channel there are always alternative 'surrogate' threads and channels that run in parallel. The constant flow of incoming variations produces significant effects in the process: it re-introduces characteristics that seemed useless or even harmful in the short term, but may be helpful in the long term. The peculiar retrieval mechanism expands the time horizon of the process: features that were dropped are not definitely discarded, but stay around for a while, and may be eventually re-used even after a long time. This keeps the system from converging too fast on possibly suboptimal configurations. Short-term efficiency is perhaps lower, to the benefit of long-term efficiency. Slow convergence,

modularity and multiple checkpoints also prevent too many errors to be made and the diffusion of errors throughout the software.

Besides novelty, further aspects of variation are recombination and *bricolage*. Most software development work consists in re-use and re-working of available components and modules that are variously re-assembled and re-combined (Ratto, 2003). The new is mixed with the old. The licensing agreement allows programmers and users to incorporate in their software products pieces and components of old products, thus inseminating products with elements of other products (Demil and Lecocq, 2003). This 'insemination" mechanism needs to be accounted for together with recombination as a way of creating and disseminating knowledge.

In spite of the potentially high connectivity, there is not such thing as all-to-all communication in open-source software projects, as the popular saying goes. Networking happens but is based on local interactions. Making an inquiry or posting a patch on the mailing list is a one-to-all communication, but only a selected few, if any, will pick up the message and respond accordingly. The data reveal that the overall interactive system is made of a network of small interacting clusters of selected agents, which coalesce, perform some task and dissemble in very short cycles. Even if in principle anybody can hook to a thread, enter the conversation and join the activity around a programming task, conversations take place among selected few. Individual agents and thread initiators migrate from thread to thread and from cluster to cluster. Clusters are ever shifting and reconfiguring, recombining tasks, activities as well as participants. Threads are minimal structures, both social and inter-temporal. We found a majority of short-term high intensity threads together with a few long-term low intensity threads.

The system has a pulsating behaviour, with periods of low intensity activity, where presumably basic routine work and communication take place, broken by sudden and short bursts of intense activity that can last one or a few days, where common attention focuses on some more engaging and urgent problem solving or design task (see again Figure 5). As we have said above, threads signal activity evolving in rapid feedback cycles of production – communication – testing – updating. The short messaging cycles generate alertness and drive attention. The rhythm of announcements, questions, patches, updates, bug notices in the mailing lists is so quick that participants are always alert and attentive to what will come next. That gives a local focus to a bunch of participants who are always on the move to do some monitoring, repairing, and communicating. The local, temporary order provided by threads has a critical inter-

temporal dimension, too: the next-next-next sequences in the chaining of communications are minimal inter-temporal structures connecting the present to the past and the future. They facilitate the participants' backward-forward travelling in time through documentation and their retrospective and prospective sense-making (Weick 1995).

5.2 Ecologies of knowledge

Several striking features should be highlighted in the evolutionary process that we have tried to reconstruct above. One is the amount of diversity that the system can handle without explicit organizational mechanisms for control and governance. Open-source projects are intricate webs connecting agents and communications, artefacts and tools, resources, problems and solutions that co-exist and interact in the internet-based programming environment. This apparently chaotic diversity becomes a powerful resource for knowledge making and innovation (Brown and Duguid, 1991). A second feature is the maintenance of a dynamic balance between the opposing requirements of stability and variety, conservation and innovation. Variety facilitates flexibility, and is counterbalanced by stabilizing mechanisms such as software modularity, communication templates and documentation protocols embedded in the technology, which enforce predictable behavioural moves. As a result the project is enough 'unstructured' and chaotic to avoid rigidity and non-adaptiveness, and at the same time structured and 'orderly enough' to ensure stability and performance, so as to be called a project at all. Large-scale projects such as LINUX and APACHE are evolutionary processes, ever shifting and drifting, and are not literally 'managed', in the sense of implementing a pre-defined plan. They can only be steered locally and reactively (Iannacci, 2002). The direction and the balance do not come from *ex ante* or centrally planned design, but rather emerge out of unplanned, decentralized interactions. Even in the case of LINUX project, in spite of the apparently dominant role of its initiator and its recent more hierarchical turn, maintainers can decide only about what the distributed activities of the many bring to their computer desks. A third feature is that the process oddly combines slow global convergence on the one hand and, on the other hand, short and fast local activity cycles, as found in the mailing lists. How can that be? A plausible answer is that the time pacing of development work is fast at the local level but the overall effect, because the system explores and processes so many variations and possibilities, is slow convergence at the global level. These features give open-source software projects superior evolutionary advantage over traditional in-house, corporate-based software production, and equip them with more effective mechanisms for exploiting highly distributed knowledge resources.

Our reconstruction of the pattern of distribution of activities and resources through the mailing lists leads to conceive open-source software projects as ecologies of knowledge. Rather than in a population dynamics perspective (Hannan and Friedman 1989; Healy and Schussman 2003), we use the notion of ecology with reference to the dynamic interaction of multiple heterogeneous elements and relationships, to their competitive or cooperative co-existence, and to the delicate equilibrium that exists between them. Following Bateson (1972), for us 'ecology' designates the mix and variety of elements that characterize the activity systems and the practices of open-source development as an evolving domain of practical knowledge and expertise. An ecological perspective captures the set of opposed but complementary features characterizing large-scale open-source software projects: variability versus homogeneity, competition versus cooperation, equilibrium versus reproduction, diversity versus standardization, recombination versus blueprint design. These dichotomies belong to the dynamics of knowledge making processes in open-source software projects and express the tension between innovation and conservation typical of complex evolutionary systems (March, 1991; Baum and Singh, 1994; Aldrich, 1999). Such 'ecological' character of open-source development gives a special quality to knowledge creation and dissemination. Open-source development activities enact a richly textured knowledge-intensive environment where multiple agents entertain loose cognitive and practical transactions with an array of artefacts and tools.

In an ecological perspective, knowledge is created by leveraging the scattered and occasional contributions of many small agents. Even if in the data we find that only around 10% of the developers actually build code, while the remaining 90% do apparently menial programming and reporting jobs at the project's periphery, nevertheless the work of the core developers (and the project itself) thrives on such a large pool of distributed knowledge resources. In other words the critical knowledge-making mechanism lies in the larger web of agents, interactions, artefacts and resources. A large open-source software project, when it reaches a critical mass, works itself as a giant decentralized mechanism for generating and distributing knowledge. The idea of ecology applied to knowledge processes suggests that whatever we call 'knowledge' in open-source software projects is the evolving outcome of the processual interplay of multiple contributions (Bateson, 1972; Sindig-Larsen,1987; Anderson and Laird, 1988). Knowledge comes out of *bricolage*, in which a lot of creative recombination and recycling of pre-existing materials takes place (Lanzara, 1999; Ciborra, 2002). New knowledge hardly emerges in frozen environments but more easily springs out of diversity and surprise, which can only occur in loosely integrated

systems where is room for controversies and multiple views. ‘Ecology’ also suggests that knowledge is not a ‘thing’ that can be purposefully managed or manufactured, but an evolving ‘complex’ that can only be fed and cultivated, kept in balance or locally innovated (Bateson, 1972; Hanseth, 1996; Blackler, 1995; Engeström, 1987; Swan and Scarbrough, 2001). No ‘knowledge system’ is up for grabs as a whole, but can only be peripherally and locally updated. In the end, this seems to be the major lesson taught by open-source software projects.

6. Concluding remarks

In this paper, using data from the developer mailing lists of two open-source software projects, we have illustrated how in open-source software projects processes of knowledge making and sharing are supported by dense social interaction and by the peculiar organizing features inscribed in technological artefacts. Discussion threads support the circulation of project-related knowledge by recording both the log of the development process and the social history of the project. The mailing lists allow for potentially unrestricted access to discussion and at the same time reveal the highly selective structure of the communication of a high number of programmers discussing a common development theme. Although in our study we have only examined a specific type of artefact, in large open-source software projects there are many of them, all playing with the delicate balance between variety and stability, innovation and conservation of knowledge. The evolving knowledge contents of the discussion threads teach us that knowledge in an open-source software environment is never a final, bounded product, but always in the making: it may perhaps reach temporary stability, but can always be subject to variation and recombination in the next round of programming and conversations. Software-based artefacts such as the developer mailing lists reflect and support such evolutionary process. Though transient and ever evolving, artefacts are critical for knowledge making and sharing in open-source software projects, as well as in other collective endeavours. Indeed, creation and dissemination do not come in a staged sequence in open-source software projects; instead they are simultaneous and closely interwoven, and they feed one another. Finally, based on our study, we submit that open-source software projects are an interesting field to study the place of artefacts in knowledge processes. A deeper appreciation of the role of material and virtual artefacts in knowledge making and sharing can help us to throw new light on the nature of knowledge itself in a variety of fields.

References

Akrich, M. and B. Latour (1992), A Summary of a Convenient Vocabulary for the Semiotics of Human and Nonhuman Assemblies, in: Bijker, W. and J. Law (eds., 1992), Shaping Technolgy, Building Society: Studies in Sociotechnical Change, Cambridge, Mass: MIT Press, pp. 259-264

Aldrich, H. (1999), Organizations Evolving, Thousands Oaks, CA: Sage

Anderson, M. and Laird, C. (1988), Evolution and Development of Knowledge Environments, Paper presented at the International Conference on Culture, Language and Artificial Intelligence, Stockholm, May 31 – June 3, 1988

Barley, S.R. and P.S. Tolbert (1997), Institutionalization and structuration: Studying the links between action and institution, in: Organization Studies 18 (1), pp. 93-117

Bateson, G. (1972), Steps to an Ecology of Mind, New York: Ballantine

Baum, J.A.C. and Singh, J.V. (eds.; 1994), The Evolutionary Dynamics of Organizations, New York: Oxford University Press

Blackler, F. (1995), Knowledge, knowledge work and organizations: an overview and interpretation, in: Organization Studies, 16, 6, pp. 1021-1046

Brown, J.S. and P. Duguid (1991), Organizational learning and communities of practice: Toward a unified view of working, learning, and innovation, in: Organization Science, 2, pp. 40-57

Ciborra, C.U. (2002), The Labyrinths of Information: Challenging the Wisdom of Systems, Oxford University Press, Oxford

Ciborra, C.U. and O. Hanseth (1998), From tool to Gestell, in: Information, Technology, and People, 11 (4), pp. 305-327

Demil and Lecocq (2003), Neither Market nor Hierarchy or Network: The Emerging Bazaar Governance, http://opensource.mit.edu

Eisenhardt, K.M. and Santos, F.M. (2002), Knowledge-Based View: A New Theory of Strategy?, in: Pettegrew, A., Thomas, H., and Whittington, R. (eds.), Handbook of Strategy and Management, London: Sage, pp. 139-164

Engeström, Y. (1997), Learning by Expanding: An Activity Theoretical Approach to Developmental Work Research, Helsinki: Orienta Konsultit

Eveland, J.D. and T.K. Bikson (1988), Work Group Structures and Computer Support: A Field Experiment, in: ACM Transactions on Information Systems, Vol. 6, No. 4, October 1988, pp. 354-379

Goffman, E. (1983), The Interaction Order, in: American Sociological Review 48 (1983), pp. 1-17

Hannan, M. and Freeman (1989), Organizational Ecology, Cambridge: Harvard University Press.

Hanseth, O. (1996), Information Technology as Infrastructure, Ph.D. Dissertation, Department of Informatics, Goteborg Universitet, Report 10, November 1996

Healy, K. and A. Schussman (2003), The Ecology of Open-Source Software Development, Working Paper, Department of Sociology, University of Arizona, Social Sciences, 2003

Iannacci, F. (2002), The Social Epistemology of Open-Source Networks, Working Paper, Department of Information Systems, London School of Economics and Political Science, 2002, pp.1-28

Joerges, B. and B. Czarniawska (1998), The Question of Technology, or How Organizations Inscribe the World, in: Organization Studies, 1998, 19(3), pp. 363-385

Kling, R. and W. Scacchi (1982), The Web of Computing: Computer Technology as Social Organization, in: Advances in Computers, 21, pp. 1-90

Kogut, B. and A. Metiu (2001), Open-source Software Development and Distributed Innovation, in: Oxford Review of Economic Policy, 17 (2), pp. 248-264

Kollock, P. (1999), The Economies of Online Cooperation: Gifts and Public Goods in Cyberspace, in: M. Smith and P. Kollock (eds., 1999), Communities in Cyberspace, London: Routledge

Kuwabara, K. (2000), Linux: A bazaar at the edge of chaos, in: First Monday, 5 (3), accessible at: http://firstmonday.org/issues/issue5_3/kuwabara/

Lanzara, G.F. (1999), Between transient constructs and persistent structures: designing systems in action, in: Journal of Strategic Information Systems 8 (1999), pp. 331-349

Latour, B. (1992), Technology is society made durable, in: Law, J. (ed., 1992), Sociology of monsters: Essays on power, technology and domination, London: Routledge, pp. 103-131

Latour, B. and S. Wolgar (1979), Laboratory life: The construction of scientific facts, Princeton: Princeton University Press

Lee, G.K. and R.E. Cole (2000), The Linux Kernel Development as a Model of Knowledge Creation, Working Paper, Haas School of Business, University of California, Berkeley

Lerner, J. und Tirole, J. (2001), The Open-source Movement: Key Research Questions, European Economic Review, Vol. 46 (2001), S. 819-826

Luhmann, N. (1975), Interaktion, Organisation, Gesellschaft, in: Luhmann, N. (ed., 1975), Soziologische Aufklärung II, Opladen, S. 9-20

Luhmann, N. (1995): Social Systems, Stanford: Stanford University Press

March, J.G. (1991), Exploration and exploitation in organizational learning, in: Organization Science, 2, pp. 71-87

Metiu, A. and B. Kogut (2001), Distributed Knowledge and the Global Organization of Software Development, Working Paper, accessible at: www.opensource.org

Morner, M. (2003), The Emergence of Open-Source Software Projects: How to Stabilize Self-Organizing Processes in Emergent Systems, in: Hernes, T. und Bakken, T. (eds.; 2003), Autopoietic Organization Theory: Drawing on Niklas Luhmann's Social System Perspective, Abstrakt Forlag, Oslo

Neff, G. and D. Stark (2002), Permanently Beta: Responsive Organization in the Internet Era, forthcoming in: P.E.N. Howard and S. Jones (eds., 2002), The Internet and American Life, Thousands Oaks, CA: Sage, 2003

Orlikowski, W.J. (1992), The Duality of Technology: Rethinking the Concept of Technology in Organizations, in: Organization Science, 3, 3, S. 398-427

Patriotta, G. (2003), Sensemaking on the shopfloor: Narratives of knowledge in organizations, in: Journal of Management Studies, Vol. 40, No. 2, pp. 349-375

Patriotta, G. and G.F. Lanzara (2003), The Inscription of Agency into Institutions, Paper presented at EGOS Workshop 'Institutional Change', EGOS Colloquium, Copenhagen, July 2003

Ratto M. (2003), Re-working by the Linus Kernel developers, http://opensource.mit.edu.

Raymond, E.S. (2001), The Cathedral and the Bazaar: Musings on Linux and Open-source from an Accidental Revolutionary, Sebastapol, CA: O'Reilly and Associates

Scacchi, W. (2001), Understanding the Requirements for Developing Open-source Software Systems, accepted for publication with revisions in: IEE Proceedings – Software, Paper No. 29840, December 2001

Schelling, T.C. (1960), The Strategy of Conflict, Harvard University Press

Simon, H.A. (1969), The Sciences of the Artificial, Cambridge: MIT Press

Sindig-Larsen, H. (1987), Artificial Intelligence and the Ecology of Knowledge. Some Background Ideas for the Programme Committee of the Conference 'Culture, Language and Artificial Intelligence', Stockholm, 1988

Stallman, Richard (1998): The GNU-Project, accessible at http://www.gnu.org/gnu/the-gnu-project.html, 1998 (edited 27th November 2001)

Strauss, A. and J. Corbin (1990), Basics of qualitative research, Thousands Oaks, CA: Sage

Swan, J. and H. Scarbrough (2001), Knowledge Management: Concepts and Controversies, in: Journal of Management Studies, Special Issue on Knowledge Management, 38, 7, pp. 913-921

Tsoukas, H. 81996) The firm as a distributed knowledge system: a constructionist approach. Strategic Management Journal, 17: 11-25.

Von Hippel, E. (2001), Innovation by User Communities: Learning from Open-source Software, in: MIT Sloan Management Review, Vol. 42 (2001), pp. 82-86

Von Krogh, G., S. Spaeth, and K.R. Lakhani (2003), Community, Joining, and Specialization in Open-source Software Innovation: A Case Study, forthcoming in: Research Policy Special Issue on Open-source Software Development (2003)

Weick, K. E. (1995), Sense-making in organizations, Thousands Oaks, CA: Sage

Zipf, G.K (1949), Human Behavior and the Principle of Least Effort, Reading, MA: Addison-Wesley Publ. Co.