_____

Faculty of Engineering

Faculty Publications

_____

Herding cats in a FOSS ecosystem: a tale of communication and coordination for release management

Germán Poo-Caamaño, Eric Knauss, Leif Singer and Daniel M. German

August 2017

_____

CrossMark

Herding cats in a FOSS ecosystem: a tale of communication and coordination for release management

Germán Poo-Caamaño[1*] iD, Eric Knauss[2], Leif Singer[1] and Daniel M. German[1]

Abstract

Release management in large-scale software development projects requires significant communication and coordination. It is particularly challenging in Free and Open Source Software (FOSS) ecosystems, in which hundreds of loosely connected developers and their projects are coordinated to release software to a schedule. To better understand this process and its challenges, we analyzed over two and half years of communication in the *GNOME* ecosystem and studied developers' interactions. Through a case study, we cataloged communication channels, determined the main channel from which we categorized high level communication and coordination activities spanning five releases, and triangulated our results by interviewing ten key developers. We found that a release schedule, influence (instead of direct control), and diversity are the main factors that positively impact the release process in the GNOME ecosystem. We report a set of lessons learned that encapsulates our understanding of how the Release Management process function in a FOSS ecosystem, we learned that: (1) ensure that the release team follows the main communication channels used by developers, (2) provide a common place for coordination for an ecosystem, (3) consider including both good technical and social skills in a release team, (4) aim for a diverse release team, (5) based on lack of power, lobbying and consensus based management must be followed, (6) help the release team in the coordination process with a well defined schedule, and (7) release team work is different from regular software work. Our results can help organizations build better large-scale teams and show that research focused on individual projects might miss important parts of the picture.

Keywords: Release management, Software ecosystem, Empirical study

"We need to communicate to make sure ... we have a product altogether that works. That components are well integrated with each other ... we don't consider GNOME a random set of tools that are totally separated from each other, we want [them] to work well [together]."
—A GNOME release team member

1 Introduction

Releasing a single software product is already challenging, but consider the challenges of releasing a complex product that consists of a multitude of independent software products. Each of these individual software products is developed autonomously, with distributed teams of developers, different motivations, many of them working as volunteers. And yet, most of the time, the complex product with all its individual pieces is released on time and in high quality. The developers of each of these pieces must communicate and coordinate effectively throughout an ecosystem of interrelated software products to achieve the goal of releasing a cohesive product.

Release management in such an ecosystem relates to both technical and social aspects of software engineering [15, 59] in a highly distributed setting [27, 30] and better understanding of this phenomenon is important both for closed and open source development.

Software development is more than writing code, it involves a set of technical and social aspects that must be taken in consideration [15, 59].

*Correspondence: gpoo@uvic.ca
[1]Department of Computer Science, University of Victoria, 3800 Finnerty Road, Victoria, British Columbia, Canada
Full list of author information is available at the end of the article

Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 2 of 24

Especially in distributed settings, the software development process requires additional work to overcome different strategic and cultural views to design, implement, and test software [27, 30]. Empirical studies in industrial settings report that *"cross-site communication and coordination issues are critical to achieving speed in multi-site development"* [30].

Yet, communication and coordination is challenging in distributed teams, and since such distributed teams is the nature of many FOSS projects [53] they offer a unique opportunity to further our understanding of release management in ecosystems.

FOSS projects can encompass a variety of methods or process es for developing software, which can vary according to the type of governance model they have [5]. The purpose of FOSS governance is three fold: to solve collective action dilemmas, to solve development coordination problems, and to create a climate for contributors [48]. Projects with a strong leadership may not require to reach consensus in the decision making process.

In solo projects as well as projects lead by a *benevolent dictator* (for example, Linux, and Perl) or by a corporation (for example, MySQL and BerkeleyDB) the direction and decision making process are clear; whereas in community and Foundation-based projects, the decisions are usually reached via consensus [4]. GNOME is a project governed by a Foundation, and it needs to find different ways to align all projects and developers. Furthermore, GNOME is a FOSS ecosystem, where the developers and leaders of multiple independent projects must reach consensus to decide what features to deliver in a common integrated product, when to deliver it, and how to reach the minimum acceptable quality to deliver it.

A FOSS ecosystem is a set of independent, interrelated FOSS applications that operate together to deliver a common user experience. Besides GNOME, other examples of FOSS ecosystems include Linux distributions (such as Debian), or the R ecosystem (R language, libraries and tools). Because in a FOSS ecosystem a release comprises many different independent applications, the release management of the ecosystem can be significantly more difficult than any of its applications alone. Release managers need to coordinate the goals and schedules of multiple teams to deliver, from the point of view of the user, one single release.

Previous research on communication and coordination in software ecosystems has focused in a temporal analysis of information flows [37], and then obtained a structural map about flows between actors [38]. To our knowledge, the requirements and challenges that release managers face in software ecosystems have not been explored, and little is known about how FOSS ecosystems conduct release management.

In this study, we empirically investigate in the complex phenomenon of the GNOME FOSS ecosystem by employing three theories that provide a lens for our inquiry: the media richness theory [14], the channel expansion theory [10], and the shared understanding theory [1]. We rely on these three theories to obtain a holistic and complete account of communication and coordination for release management.

In their *media richness theory*, Daft and Lengel [14] argue that organizations process information to reduce uncertainty and ambiguity, for which the communication channels and organizational structure play an important role. The communication channels vary in their capacity for providing richer or leaner information.

Rich communication channels, such as face-to-face and video interactions, enable immediate feedback, cross-check of information, and provide additional cues, such as body language, tone, and message content in natural language. In contrast, leaner communication channels, such as email or instant messaging, lack the ability of conveying nonverbal cues, and the feedback is limited [41]. In addition, a second source of uncertainty is produced by the need of integration between multiple teams or projects within an ecosystem: "people come to a problem with different experience, cognitive elements, goals, values, and priorities" [14]. FOSS ecosystems may be limited to use certain communication channels, and the diversity of projects and developers' background may increase the difficulties to reach consensus, motivating our first two research questions:

RQ_1 What are the communication channels used for release management?

RQ_2 How do developers communicate and coordinate for release management?

The *channel expansion theory*, proposed by Carlson and Zmud [10], presumes that as individuals gain experience in the use of a channel, they are able to use the media channel more effectively; such experience involves the use of the channel itself, the organizational context, the message topic, and the understanding of how their co-participants communicate. Furthermore, according to Dennis and Valacich [16], the development of standards and norms increases with the experience that a group gains, and as a result such a group may also experience an improvement in the interplay and the tasks they perform over time. This theory may or may not apply on how a successful FOSS ecosystem releases software on time and indicates a need to further understand the usage of channels for release management.

Finally, the *theory of shared understanding* proposed by Aranda [1], argues that the formalization of a process in a software project helps reduce the overall complexity,

Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 3 of 24

because once a process is formalized, it facilitates that the stakeholders familiarize with the process itself rather than the details of each part of the organization. As a consequence, the stakeholders can make assumptions on parts of the organizations that are unfamiliar to them, and the organizational complexity is reduced.

Motivated by the channel expansion and shared understanding theories, we aim to understand the key actors that help to align a distributed team of mostly volunteers, and to understand what kind of tasks they have that enable the coordination across the ecosystem.

RQ_3 Who are the key actors in the release management process?

RQ_4 What are the release management tasks in a FOSS ecosystem?

Finally, motivated by all three theories, and to understand what could be missing in any or all of them when we study a FOSS ecosystem, we investigate research question five:

RQ_5 What are the challenges that release managers face in a FOSS ecosystem?

To answer these research questions, we studied how the release management is done in the GNOME project.

We chose GNOME because it is a large and mature software ecosystem [49], it has been studied before [22, 36, 39, 46, 47, 67, 79], its official release is a single product comprised of many independent and distributed projects, and *more important, it has a successful and stable release schedule*: a new GNOME release is issued every six months. We studied the high level communication of the release management process across five releases.

This research will improve the way of working in GNOME and similar FOSS ecosystems.

While other software development efforts might be able to profit from our insights as well, our research does not allow generalization of results beyond the specific scope of our study, i.e. the release management within the GNOME FOSS ecosystem.

The contribution of this paper is threefold:

(i) An empirical study of Release Management in a FOSS ecosystem: This empirical study deepens our understanding of the release management practices in a FOSS ecosystem. Empirical studies aim to investigate complex real life issues where analytical research might not be enough [66]. In particular, empirical software engineering aims to understand the software engineering discipline by treating software engineering as an empirical science [60]. In this sense, our study is contributing empirical data to the body of knowledge about release management in FOSS ecosystems, which in turn will improve the way of working in GNOME and

similar FOSS ecosystems and facilitate future research in this area.

(ii) A set of lessons learned: Based on the empirical studies, we report a set of lessons learned that encapsulates our understanding of how the Release Management process function in FOSS ecosystems. We learned that: (1) ensure that the release team follows the main communication channels used by developers, (2) provide a common place for coordination for an ecosystem, (3) consider including both good technical and social skills in a release team, (4) aim for a diverse release team, (5) based on lack of power, lobbying and consensus based management must be followed, (6) help the release team in the coordination process with a well defined schedule, and (7) release team work is different from regular software work.

(iii) An exception to the richness media theory in a FOSS ecosystem: In a FOSS ecosystem, with the variety of cultural and technical backgrounds of its members, the main communication channel for coordination is asynchronous (usually mailing). Email is egalitarian because it allows contributors with different levels of English skills to participate in equal terms (something that it is hard to achieve in synchronous channels, where contributors with better language skills can dominate a discussion). In an overview of the research literature, we did not find references to language barriers in the use of communication channels in software development, perhaps because they did not study teams with the same level of variability of national origins as the FOSS ecosystem we studied.

2 Background
A software ecosystem is a set of software projects that evolve together, share infrastructure, and are themselves part of a larger software project [45, 46].

In previous work [38] we identified the existence of three major streams in software ecosystems in research: (1) software platforms and architecture, which includes modelling and architecture such as software evolution, software architecture, and software development as product lines [8] (2) business and managerial perspectives [34, 35], and (3) FOSS ecosystems [45, 68]. In addition, an ecosystem can be studied in-the-large or in-the-small [49]. That is, the interactions with external actors, or the inner ones, respectively. Our research complements these works and is focused in the inner parts of a FOSS ecosystem, that is, an ecosystem "in-the-small".

Goeminne and Mens first explored potential research questions to assess the quality of FOSS projects and that could lead to an improvement of the software development process [24]. A further study focused on the social aspects in FOSS ecosystems; in particular, the intersection of roles among developers and their activities. Developers might play multiple roles in a FOSS ecosystem, each role involves a set of activities and interactions with

Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 4 of 24

other developers that are needed to articulate the tasks in software development [49].

This paper aims to further the understanding of communication and coordination in a software ecosystem with respect to release management. We studied the enabling factors to deliver a product in a FOSS ecosystem with many individual projects. To this end, we considered the organizational structure, its communication channels, and the interaction between developers of different projects towards a common goal.

2.1 Social aspects and communication channels

FOSS development teams use multiple communication channels. There is a prevalence of certain channels over others, depending on the projects and the resources available to them, and because usually FOSS projects are developed by groups of people distributed across the globe. FOSS projects might use different communications channels, among them mailing lists and IRC are the most frequently used [21, 23, 25]. Mailing lists are used as public forums for asynchronous communication whereas IRC is used as instant messaging for synchronous communication. Both communication channels correspond to leaner channels because of lack the ability of conveying nonverbal cues and the limited feedback [41].

Leaner communication channels are effective to process standard data and well understood messages, however, they may require rules and procedures. The complexity in the communication increases when there are multiple teams or projects within an ecosystem that require coordination, and who may have multiple conflicting interpretations of the same piece of information. High ambiguity in an organization means confusion and lack of understanding [14].

When the differentiation between teams and projects is small, but with high interdependence, then the coordination can rely on leaner communication channels because the ambiguity is low. When the differences are high, then a communication channel with high richness can help reduce ambiguity, which may be challenging for a FOSS ecosystem to use. The frequency of communication will depend on the interdependence between them. The higher the dependency, the higher the coordination needs.

Michlmayr and Fitzgerald [51] reported that the parallel and independent nature of FOSS development reduces the amount of active coordination needed in such projects. Yet, it is important to synchronize teams and projects regularly to establish awareness of changes that may create conflict.

From a cognitive point of view, the media richness of communication channels is not enough to get the information understood by the participants. The participants must be motivated to process a message and have the ability to process it [63]. Richer communication channels induce a higher motivation, but the receiver requires more abilities to process such information because there is more information to process; and richer communication channels are also synchronous, giving the receiver less time to process the message. The opposite happens with leaner communication channels: they decrease the motivation but increase the ability to process a message. This is what Robert and Dennis [63] call *"richness media paradox"* because the rich media can simultaneously improve and impair the communication.

2.2 Release management

We studied the factors that allow a distributed FOSS ecosystem to deliver a product that involves coordination among many individual projects. To this end, we considered the organizational structure of the ecosystem, its communication channels, and the interaction between developers of different projects towards a common goal.

Michlmayr [50] studied the impact of schedules on release management in FOSS projects, with an emphasis on time-based schedules in seven projects. He characterized the challenges in release management that FOSS projects face and the practices they use to cope with them. Building on top of these contributions, this paper considers the communication needs to coordinate multiple teams and projects in software ecosystems with focus on release management.

To overcome the challenge imposed by the apparent informality in the FOSS development, Erenkrantz [19] examined the release management in three FOSS projects and proposed a taxonomy for identifying common properties to compare the release management in FOSS projects. The properties evaluated were: release authority (who decides the release content), versioning (what is the scheme to name the release versions), pre-release testing, approval of releases (who approves a the software is ready to be released), distribution (how the software is distributed), and formats (in which formats the software is released). We did not find evidence of other studies using this taxonomy.

2.3 Background on selected case: the GNOME ecosystem

The GNOME Project was started in 1997 by Miguel de Icaza and Federico Mena-Quintero to create a collection of libraries and applications that could make Linux a viable alternative in the desktop. The main components of GNOME are: an easy-to-use GUI environment, a suite of applications for general use (for example, email client, web browser, music player), and a collection of tools and libraries to develop applications for GNOME [22]. All of these components are highly integrated, resulting in a common product: the GNOME Desktop.

Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 5 of 24

2.3.1 Organization of the GNOME ecosystem

From an organizational point of view, GNOME is a federation of projects in which each project acts independently of the rest and has its own internal organization, yet they collaborate to create the GNOME Desktop.

To organize around these highly integrated components a non-profit organization was created in 2000: the GNOME Foundation [22]. According to official statements, the goals of the GNOME Foundation are: (1) to create a legal entity around GNOME (2) to manage the infrastructure to develop and deploy GNOME, and (3) to coordinate releases.

The GNOME Foundation does not have direct power over the individual projects or developers, most of whom are either volunteers or paid employees of companies. Instead, it aims to fulfill its goals by creating consensus and policies. The GNOME Foundation is headed by a Board of Directors that is democratically elected by the developers who are Foundation members. Any developer who has made a non-trivial contribution to GNOME can apply to become a Foundation member, a membership that has to be renewed every two years [73]. The Charter of the GNOME Foundation states that one of the first duties of the Board of Directors was to appoint a Release Management Team [74].

The GNOME Foundation's Board of Directors receives input from an Advisory Board. The Advisory Board is comprised of members of companies who directly fund GNOME. The Board of Directors delegates administration tasks to an executive director and technical issues to the release team.

2.3.2 Cross-cutting teams

Cross-cutting teams are teams who contribute to multiple projects within the GNOME ecosystem; they provide specialized expertise on areas that are common across all projects, and that usually individual projects may lack. Examples of such teams are the *Translation Team*, the *Accessibility Team*, as well as teams for *Quality Assurance* and *Documentation*. Cross-cutting teams are responsible for supporting the activities of project teams and the overall success of GNOME as an integrated environment. For instance, the Accessibility Team makes sure that every application in GNOME can be used by users with disabilities; the Accessibility Team develops libraries to enable applications to interact properly with screen readers, braille keyboards, and other similar devices, and simultaneously, the Accessibility Team works in every other library and application in GNOME to use such APIs. The Translator Team makes sure the applications, with their respective documentation, are available in multiple languages, and that these translations are consistent across applications; to accomplish that, the Translator Team monitor that every text available in the user interfaces

can be translatable, and maintain the tools that facilitate the work for developers and translators. Like project teams, they have their own internal structures and decision making processes, and act independently of other teams; however, their purpose is to help other projects on specific tasks.

2.3.3 Relation between projects and cross-cutting teams

In GNOME, the release management tasks are performed by the release team. This team does not have any official power over any other team or its members. However, the release team decides which projects to include—and by extension, to exclude from—the official GNOME Desktop release.

The release team decisions are expected to help in the scheduling of activities of the cross-cutting teams. For example, the Translation Team requires time without changes to the user interface to translate applications into different languages. This demand can be satisfied better at the end of a release cycle.

2.3.4 Previous studies of GNOME

The GNOME project has been widely studied. There are studies on the workload of contributors and projects in the GNOME ecosystem. Vasilescu et al. [78] determined that the workload varies depending of the type of contributor. Koch and Schneider [39], and German [22] reported that the distribution of workload with respect to file touches is left–skewed, where few developers contribute most of code. Casebolt et al. [11] compared authoring with respect to the file size, and suggested that large files are likely to be authored by one dominant contributor. Lungu et al. [46] focused on the visualization of the source code activity in the GNOME ecosystem over time, and distinguished three phases in GNOME's lifetime: introduction, growth, and maturity.

Several studies have focused on GNOME's bug database (*Bugzilla*), whose purpose have been: (1) predict the bug severity [40, 42] (2) determine quality of bug reports [2, 69], and (3) determine the efficiency of developers to address issues [44].

Studies related to communication channels have been focused in set of projects, for example, Shihab et al. [71] mined the meeting logs of the GTK+ project (a core library in the GNOME ecosystem) held on IRC (Internet Relay Chat). Pagano and Maalej [57] used LDA to explore how developers communicate through blogs. One of the outcomes was that developers usually write blog posts after an engineering activity such as committing a new feature, fixing a complex or important bug, or releasing a new version of a piece of software. Lungu et al. [47] argued that studies based on multiple projects treated individually, and not as part of an ecosystem, miss the opportunity to study the context of the projects.

Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 6 of 24

3 Study design

To answer the research questions, we used a mixed methods approach [18] that employs data collection and analysis with both quantitative and qualitative data analysis [13, 66, 82]. The study was composed of four steps: (1) we identified the main communication channel used for high level coordination between projects and teams within the ecosystem (2) we collected and cleaned the data from that channel (3) we analyzed the data collected and extracted discussion themes, and (4) we conducted interviews to triangulate our findings and obtain additional insights from developers.

3.1 Communication channel selection

To identify the communication channels used for release management, we learned about the GNOME organization by gathering and consolidating information found on its website. Two main communication channels are recommended: mailing lists and IRC. We focused on mailing lists, as they are archived and publicly available. We did not find evidence that communication over IRC was archived by GNOME, which makes its historical analysis harder, if not impossible.

3.2 Data collection and cleaning

We identified 285 mailing lists archived in the GNOME ecosystem. We searched for mailing lists used for cross-project communication and release management. We found that the release team recommends to its new team members to follow two mailing lists (*desktop-devel-list* and *release-team*) to help new release team members grasp background information about the development process within the ecosystem [76]. The subscription to the *release-team* mailing list is limited to the release team members, therefore, it is an internal mailing list and no a cross-project communication channel. Because we are interested in cross-project communication and coordination within the software ecosystem, we focused our study in the main channel that serves that purpose.

We identified the Desktop Development mailing list [75] as the main channel for information related to release management: it is where the discussion of the desktop and platform development takes place. To study the communication across several releases, we retrieved data for 32 months spanning from January 2009 to August 2011. We used MLStats [64] to split into threads the mailing list archive data sets. We chose this period because it comprises 5 release cycles, including the transition between two major releases–from the series 2.x to 3.x. The transition from the series 1.x to 2.x lead to frustrations among developers, as well as skepticism that a—back then new—time-based release would allow to pursue a new major version [50]. Therefore, the transition between major releases was compelling to study. In total, we analyzed

6947 messages (an average of 214 messages per month). These were grouped into 945 discussions with 1 to 50 participants each, and a median of 2 participants per discussion.

To associate multiple email addresses with a single individual, we used an approach similar to the cluster algorithm described by Bird et al. [6]. We created clusters of similar identities and then manually processed them. To create the clusters we used both name and email; we first normalized the names, then we looked for name similarities, name-email similarities, and email similarities. To match identities, we also collected names and email addresses from other data sources, such as commit logs and projects' metadata. Based on GNOME's account name policy [72], we merged email addresses ending in *gnome.org* that had the same user name (for example, we merged in *jhs@gnome.org* email addresses like *jhs@cvs.gnome.org*, *jhs@src.gnome.org*, and *jhs@gnome.org*).

3.3 Analysis

We followed a Grounded theory [12, 13] approach to analyze the discussions in the *desktop-devel-list* mailing list. In Grounded theory, researchers label or code openly the data to uncover themes and extract concepts. Through manual analysis we segmented the email subjects into categories and labeled them with a term, extracting themes from the discussion threads.

To code the messages we read the email subjects and associated a code to each thread. The code then represented the message's theme. Whenever the subject was unclear, we read the discussion thread in detail, and searched in other data sources (for example, wiki, websites, related bugs and source code commits referenced in the discussion) for additional clues about the topic discussed. Thus, we also considered the role in the ecosystem of the person initiating a discussion, the roles of the other participants in the discussion, the number of messages in such discussion, the number of participants in a discussion, and the time in the release cycle were the discussion occurred—from early planning to finally releasing a stable version. We used those details as follow:

Role (initiator). To know an individual's status in a project within the ecosystem, and the potential motivations to bring a topic to discuss. We assumed that the intention of a message may vary depending of the sender (*user*, *regular developer*, *project maintainer*, or *team member*).

Role (participants). To know specialties and type of discussion they became involved with. We could distinguish among people who replied to regular developers or newcomers in the mailing list, and whether

Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 7 of 24

developers would participate in familiar subjects or in broader discussions.

Number of messages. To order the discussions. Discussions with only one message (no reply) were left to the end.

Number of participants. To order the discussions. Discussions with several participants were investigated with more detail.

Release cycle time. To contextualize the discussions studied and determine discussion patterns that depended on the stage in the release cycle.

We clustered codes into categories of communication and coordination. The first author manually categorized the email subject fields, which were presented to the other authors for discussion, avoid misinterpretation, and to derive categories. Later, we validated these categories through interviews with the corresponding developers.

3.3.1 Social network analysis

To determine key developers in GNOME's release management, we conducted a social network analysis of the mailing list. In this social network, a node represents a participant in a discussion. A participant who replies an email is connected to all the previous authors who have participated in the same discussion thread sorted chronologically. As Bohn et al. [7], we assume a respondent is aware of all the previous emails in the same discussion thread. An edge between two nodes represents—undirected—communication between two developers who share some interest in the topic discussed.

We explored this social network using *Gephi* [3]. We applied the ForceAtlas algorithm [33] provided by Gephi [3]. This algorithm spatializes small-world networks with an emphasis on layouts to support analysts interpreting the graph. It pushes influential nodes to the centre and less influential ones to the border.

We determined influential nodes by calculating the degree centrality and eigenvector centrality. We calculated degree centrality to determine key participants in discussions based on the numbers of messages sent or received by a developer. The eigenvector centrality determined the influence of certain developers in the social network. Although degree centrality also measures the influence, eigenvector centrality favors nodes connected to other nodes that are themselves central within the network.

Through betweenness centrality, we determined *gatekeepers* between developers. The higher the betweenness, the higher the potential of an actor to be a gatekeeper [70].

3.4 Interviews and triangulation

The purpose of interviewing developers was twofold: first, to triangulate our findings, and second, to enrich our finding with additional insights of the development practices

and release management process. We conducted semistructured interviews with GNOME developers who had actively participated in the discussions we studied. We recruited 10 (out of the top 35 candidates) developers during GNOME's main conference, the *GUADEC*, where we performed the interviews in person. The length of the interviews varied from 26 to 93 min. All our interviews were recorded, later transcribed, and code following a Grounded theory [12, 13] approach. We added labels next to each answer in the transcriptions, in particular to determine when there were several topics combined in the same answer. Then we grouped the common themes across interviews, and manually analyzed them. The coding was performed by the first author, and reviewed and discussed with the fourth author.

The interviews consisted of three parts: (1) inquiry about roles in the project and communication channels our interviewees used (2) to comment on our findings; to probe the extend to which our findings matched their perception of their and others' communication and collaboration activities (3) to comment on specific interactions with other developers and on the circumstances in which they would feel inclined to talk with them.

First, we asked each interviewee questions of the use of communication channels as consumers and producers of information, frequency they used each channel, how and when they used each channel, and the importance they gave to each channel and to elaborate their answers. We also asked each interviewee how their roles influenced the use of certain channels, and how they take decisions when there were disagreements between developers in charge of different components in the project.

Second, we probed the extend to which our findings matched their perception of their and others' communication and collaboration activities. We presented to our interviewees our categorization of the communication in the *desktop-devel-list* mailing list, the distribution of the question types, what were their perceptions of our findings, and what we could have missed.

Finally, we used the social network to ask the interviewees about their interactions with other developers on the mailing list. To make it easier to spot the interviewees in the social network, we printed custom social network charts per developer, highlighting their interactions with others developers. In an open question style, we asked each interviewee about their interactions with other developers, and to elaborate the circumstances they would feel inclined to participate with them, their relationship if they had any, if they were aware of amount of interactions they had (for example, their importance in the social network). To enable us to engage in the interviews better, we familiarized ourselves with the type of discussions that each of these developers participated, and selected the ones we considered may bring us richer answers

Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 8 of 24

(in case we needed to remind the interviewee their interactions).

3.5 Threats to validity

In this section we discuss potential threats to the validity we identified on this case study and its results.

3.5.1 Construct validity

Construct validity refers to whether the studied parameters are relevant to the research questions, and actually measure the social constructs or concepts intended to be studied. Our analysis is based on data we extracted from public archives. We found two main communication channels in GNOME: IRC and mailing lists. We also found several secondary communication channels, such as blogs and conferences. As we focused on communication on one mailing list in our study, we might have missed some interactions that happened on other channels, such as IRC which is not archived. There could also be GNOME developers who do not participate in the mailing lists at all and instead rely on other communication channels. However, previous research suggests that the majority of discussions occur in mailing lists [6, 21, 22, 54, 55]. We also triangulated our results by interviewing key developers we identified. It is thus unlikely that our analysis missed important coordination types, patterns, strategies, or challenges.

3.5.2 Internal validity

Internal validity relates to the validity of causal inferences made by the study, and the researcher bias is a threat to the internal validity. The first author manually categorized the email subject fields, and he might have introduced subjective bias in the results. We followed Creswell's guidelines [13] for coding, which consists of abstracting common patterns in the communication process. This activity involves segmenting sentences—in this case an email's subject field—into categories and labeling them with a term. The first author extracted the topics to build the categories based on the interpretation of the subject field of each email thread. To avoid misinterpreting the actual discussions, before the coding, the first author familiarized himself with the email threads, read some of the messages to obtain more contextual information, and discussed with the other researchers in the team to soundcheck intermediate results and particular interpretations of the data. Finally, the results were triangulated by interviewing developers.

3.5.3 External validity

External validity is concerned with the extent to which it is possible to generalize the findings. In this paper, we presented a single case study, which may impose a threat to generalization of the results. However, a single study case can lead to a generalization through analytical generalization, which is performed by comparing the characteristics of a case to a possible target [20]. The case study presented can facilitate the analytical generalization and comparison with other cases.

4 Findings

In this section we present our findings structured by the respective research questions they answer. We report our findings based on the analysis of mailing list communication, social network analysis, interviews. To illustrate some findings, we provide quotations from interviews and give examples of developer viewpoints. Among similar opinions, we chose to quote only the one we considered the most representative for each case

4.1 What are the communication channels used for release management?

In our interviews, we found that the release team may monitor a variety of communication channels to have multiple sources of information that could be relevant to a release. All these communication channels can help the release team to track the development progress in GNOME. As indicated by a former release team member:

> *"[The release team] may include any input [—data source or communication channel—] when they decide."*

According to our analysis of mailing lists and follow-up interviews, however, the release team prioritizes in four of them. This is consistent with the guide for new release team members [76], which recommends participating in three mailing lists (*release-team*, *desktop-devel-list*, and *devel-announce-list*) and one IRC channel (*#release-team*).

In this section, we report the main communication channels, provide an overview of the other communication channels, and describe how they are used for release management.

4.1.1 Main communication channels

Mailing lists. In GNOME, there are internal and global mailing lists. The former are used by teams for their own purposes, the latter are used to discuss topics that concern the whole ecosystem. The release team uses an internal mailing list (*release-team*) to discuss and decide issues directly related to release management, and a global one (*desktop-devel-list*) for the whole ecosystem.

> *"If you [need] high level coordination that affect the entire project that tends to be on the mailing lists."*

Membership to the internal list is limited to the release team members, although it can receive emails from any address and the archives are publicly available.

Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 9 of 24

IRC. An interactive chat system. Similar to mailing lists, there are internal and global chat channels. The release team holds meetings once or twice per release cycle using an internal channel (*#release-team*), which is also used for discussions within the team and for developers to get quick answers on release management. For awareness of the ecosystem, the release team monitors *#gnome-hackers*.

> *"If people are already involved in working on something, IRC works very nicely for coordination."*

4.1.2 Other communication channels

Bugzilla, the web-based bug tracking system, is used to keep track of features and critical bugs for future releases [76]. The bug tracker is also used in conjunction with mailing lists and IRC to obtain awareness of issues that must be solved or require further discussion.

A *wiki* is used to maintain information of the release process, provides instructions for developers to make releases, and details of the current release schedule, such as important dates.

In GNOME, the developers' blogs are aggregated in a common location called *Planet* (*"a window into the world, work and lives of GNOME hackers and contributors"* [80]. Some release team members use them to communicate release-related decisions and to inform others about the release status, as well to monitor any concern from developers, who express their points of view regarding the project.

Face-to-face interactions occur in *conferences* and *hackfests*. During the annual conference, the release team discusses GNOME's future with developers. *Hackfests* are focused face-to-face meetings of developers to work on a specific project, feature, or release; and depending on the topic, some release team members are invited to participate to bring their perspective. Although face-to-face interactions are highly valued because of the "high bandwidth", the participation is limited only to the developers able to attend.

> In GNOME, the release team uses mailing lists and IRC as the main communication channels for coordination; for long term discussions, and for quicker decisions that involve less than four people, respectively. Regardless, the release team might use multiple channels as input to gauge their decisions, including face–to–face meetings.

4.2 How do developers communicate and coordinate for release management?

We found that developers use different communication channels, some of them specific to a particular topic or project and others for wider discussion. In the latter, discussions can be either about process management, technical issues, or both.

From our analysis of the *desktop-devel-list* mailing list, nine discussion categories emerged. Five of them are directly related to release management activities:

Request for comments. Long-term proposals that affect the entire ecosystem and require a high level of coordination. They may involve discussing the vision of the project for the next releases and beyond or major changes whose execution could take one or more releases. These discussions start at the beginning of each release cycle, and revisited during the release cycle. The release team gauges the overall sentiments. Examples: *"empathy integration with the desktop"*, *"Consolidating Core Desktop libraries"*, *"RFC: gtk-doc and gobject introspection"*.

Figure 1 shows that *requests for comments* happen through the whole development cycle in spite that these discussions are encouraged at the beginning of the cycle. The x-axis shows the milestones of the release cycle (compare Fig. 2), and the y-axis shows a box plot of the number of discussions we found in each milestone.

Proposals and discussions. Short-term proposals focused on the current release cycle and tied to a particular project, but with potential indirect impact on other projects or teams. For example, a project wanting to use a library that is external to GNOME must submit a proposal. Other projects interested in the library might support the idea or raise concerns if they are already using an alternative library. The release team may raise concerns regarding the long-term sustainability of the external library—such as development activity, availability, or the library's track record regarding security fixes. Examples: *"systemd as external dependency"*, *"Module Proposal: GNOME Shell"*, *"New proposed GnomeGoal: Add code coverage support"*.

Figure 3 shows that *proposals and discussions* happen through the whole cycle.

Announcement Notifications for developers about the status of a component or the whole project. The purpose is to raise awareness among developers and keep them engaged. Announcements include the releases of new versions, a new branch, new external dependencies, and status reports of the project goals. Examples: *"GNOME 3.0 Release Candidate (2.91.92) Released!"*, *"GNOME 3.0 Blocker Report"*.

Figure 4 shows when the *announcements* take place during the development cycle.

Schedule reminders. Specific type of *announcement* used by the release team to send periodic reminders of the release cycle's stage. The release team reminds developers to release a new version, start the period of feature
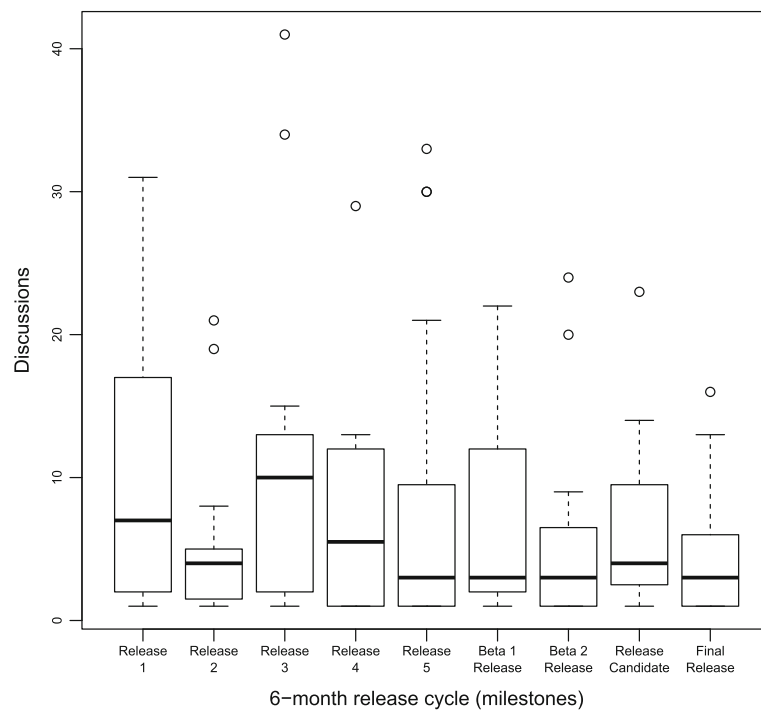
Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 10 of 24

Fig. 1 Request for comments during the release cycle

proposals, and so on. Its nature and recurrence make it worth a category by itself. Examples: *"Release Notes time!"*, *"GNOME 2.29.90 beta tarballs due"*, *"Last call for comments on module proposals"*.

 Figure 5 shows that *schedule reminders* happen through the whole cycle, however, some milestones receive more reminders than others.

Request for approval. Request to break the *freeze* period at the end of the release cycle, once the release team controls the changes (See Section 4.4). The discussion is open to everyone, but the decision is taken by the release team, the Documentation Team, or the Translation Team. These requests require a timely decision as they occur close to the release date. All decisions require at least two votes from the release team. Changes in translatable strings will also require the approval of the Documentation and Translation Teams. Changes in the user interface will also require the approval of the Documentation Team [76]. Examples: *"Hard code freeze break request*
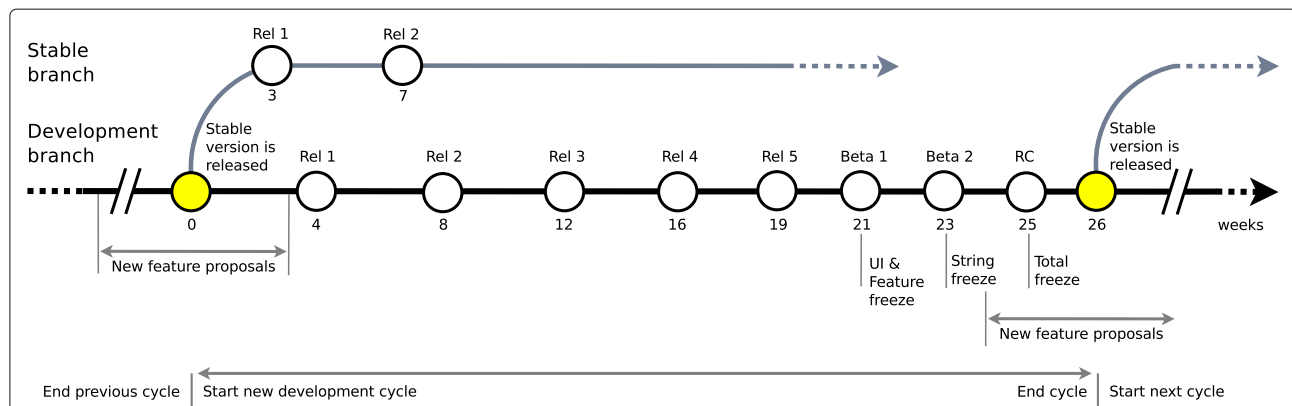

Fig. 2 GNOME six-month release schedule in weeks and its milestones. A release cycle starts immediately after a major release. The stable version is kept in a branch for bug fixes, whereas the regular development continues in the main branch. The release cycle starts with a release meeting to evaluate the last cycle, and to discuss the next cycle. The stages drive the type of communication. This illustration is based on data obtained from multiple release schedules available on the release team's wiki page and from our interviewees
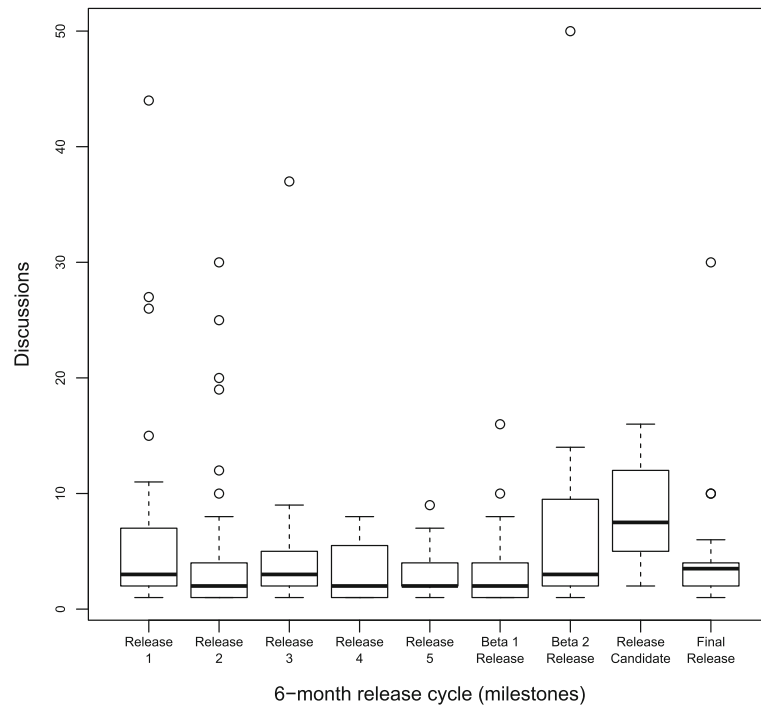
Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 11 of 24

Fig. 3 Proposals and discussions during the release cycle
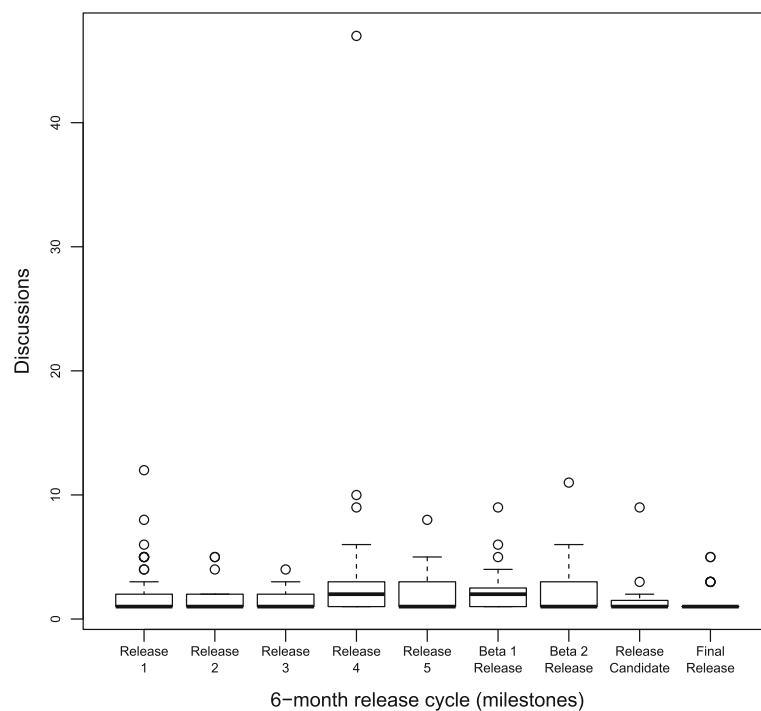
Fig. 4 Announcements during the release cycle

Poo-Caamaño *et al. Journal of Internet Services and Applications*   (2017) 8:12
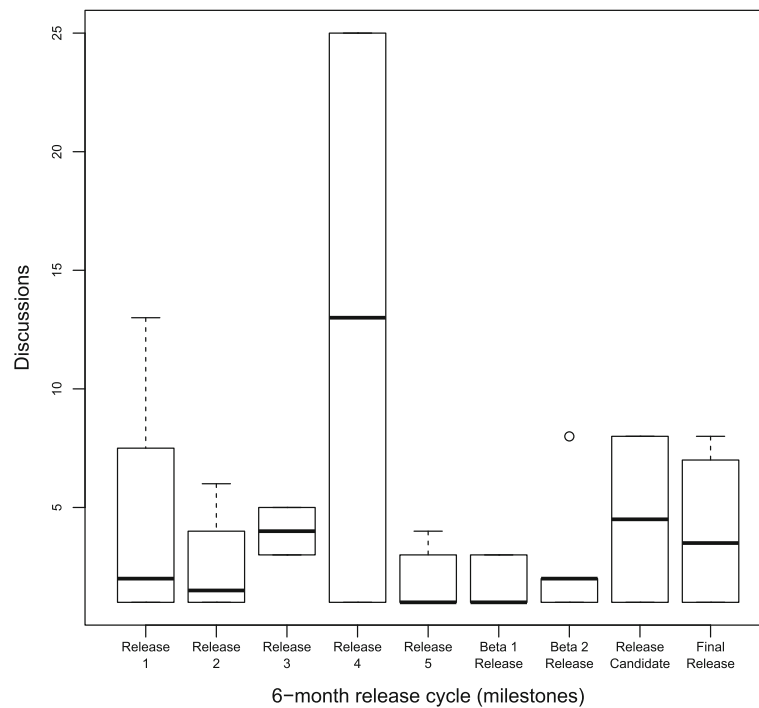
Page 12 of 24

Fig. 5 Schedule reminders during the release cycle

*for gvfs", "[Freeze break request] gtksourceview crash",
"String change in gnome-session".*

Figure 6 shows that *requests for approval* happen
at the end of the development cycle, in the period
when the release team must approve changes to the
project.

Table 1 presents the amount of discussions and
messages during the period studied. Both help balance
their importance. Although there are less *Request for
comments* and *Proposal and discussions* than *Announce-
ments*, the proportion of messages of each of them
reflects that those are the core of the discussions in the
mailing list.

We noticed that discussions started by well-known
developers attract other well-known developers, more
than discussions started by other people. Our intervie-
wees reported that they would be more inclined to partic-
ipate in a discussion started by known developers, as they
already know their expertise.

The remaining four categories are less relevant to
release management activities: *Events coordination* (spe-
cial type of announcement related to the organization
of conferences, sprints, or hackfests), *expertise seeking*
(questions on seeking others working on or in charge of
a specific part in GNOME), *knowledge seeking* (questions
from developers on specific organizational issues), and
Out of scope (any other message).

> *The release schedule of GNOME guides the type
> and timing of coordination activities discussed in the
> main communication channel. The scope of discus-
> sions span from long-term to short-term planning, and
> from the entire ecosystem to localized in particular
> projects*

4.3 Who are the key actors in the release management process?

As described in Section 3, we conducted a social net-
work analysis to determine key developers in the release
management process, which we then complemented with
interviews. Figure 7 shows such a social network. We
present the reach of six different release team mem-
bers in the same social network, whose interactions are
highlighted in each subfigure. The box in each subfig-
ure surrounds the developers (nodes) that interact directly
with each release team member. Thus, we can observe the
scope of interaction of each release team member within
the social network.

Among the three major nodes, the bigger one is the
release-team mailing list in which only release team mem-
bers participate. The *release-team* mailing list has a nor-
malized eigenvector of 1.0, which means that this node
is the most influential in the social network. In other

Poo-Caamaño *et al. Journal of Internet Services and Applications*   (2017) 8:12
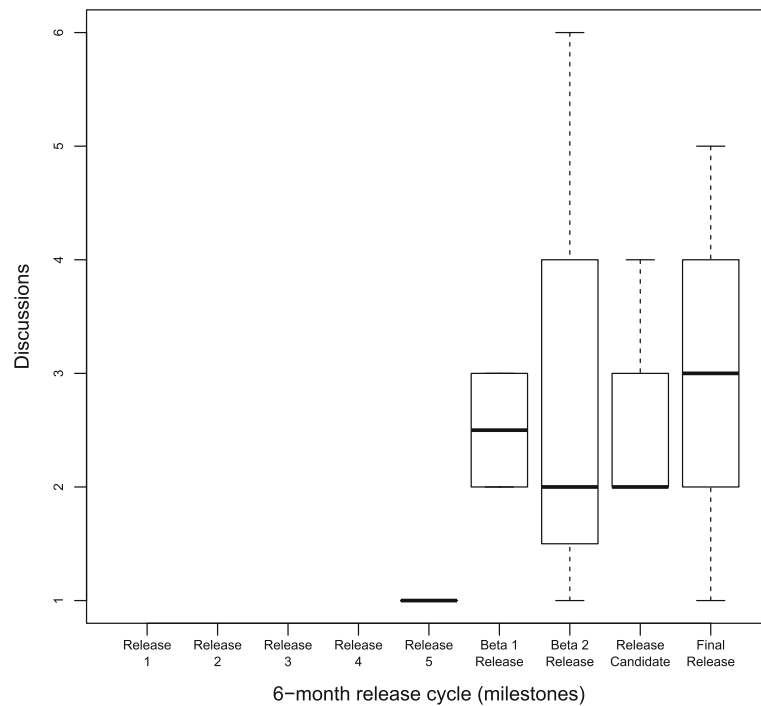
Page 13 of 24

Fig. 6 Request for approvals during the release cycle

words, the *release-team* list is usually added to discussions to make release team members aware of them. On the opposite side, its betweenness centrality is 0.0: the *release-team* list is not a gatekeeper or bridge between developers at all. As mailing lists are not associated with any person and do not send emails, this is expected. The eigenvector and betweenness centrality values for *release-team* list were the same before we pruned other mailing lists from the social network, that is, to visualize better the developers. With respect to the other two major nodes, one is a senior release team member (highlighted in more detail in Fig. 7e) and the another one a core developer.

Considering the prominence of the release team mailing list in the social network, the overall coverage of the release team members in the discussions (Fig. 7),

Table 1 Summary of discussions and messages per category

| Category	Discussions		Messages		Median of messages per	
	#	%	#	%	Discussion	Release
Discussions related to release management						
Announcement	238	25.19	740	10.65	1	413
Proposals and discussions	219	23.17	2074	29.85	4	427
Request for approval	22	2.33	83	1.19	3	18
Request for comments	181	19.15	2505	36.06	6	1192
Schedule reminders	45	4.76	236	3.40	2	12
Discussions unrelated to release management						
Events coordination	27	2.86	44	0.63	1	120
Expertise seeking	25	2.65	184	2.65	3	70
Knowledge seeking	151	15.98	764	11.00	3	971
Out of scope	37	3.92	317	4.56	2	26
Total	945	100.00	6947	100.00		
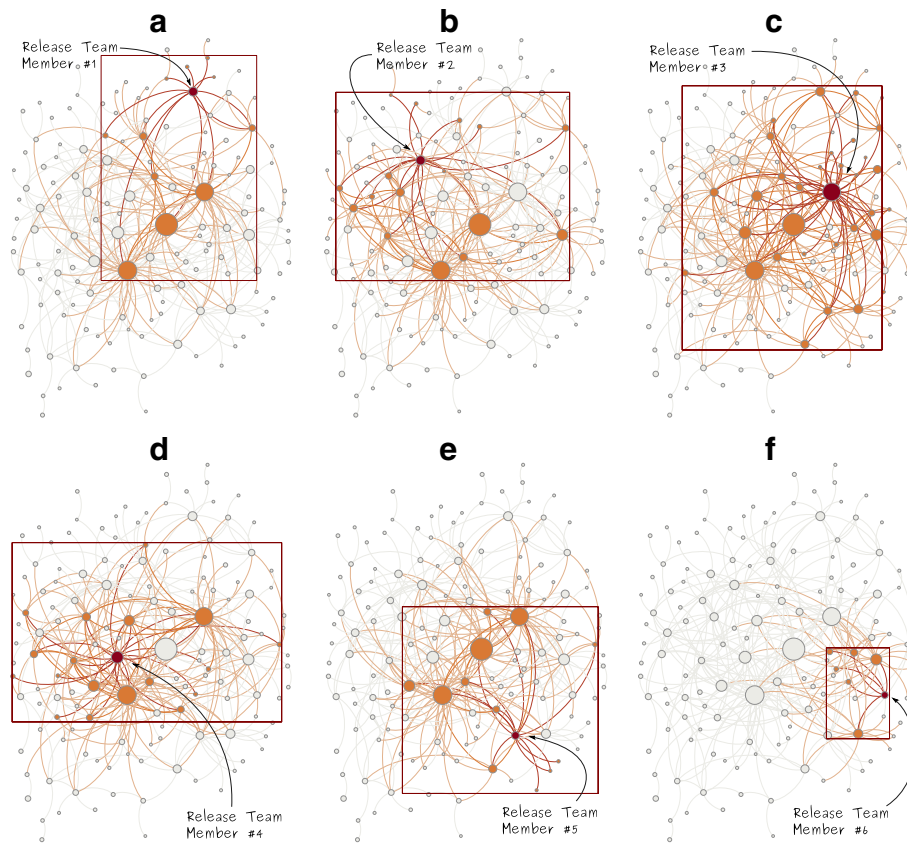
Fig. 7 Scope of interaction of six different release team members in the mailing list social network. The underlying social network is the same. Each graph highlights the direct interaction of a different release team member with other developers. Two participants are connected if one of them replies to the other. The size of the node represents the number of messages that a participant is involved. *Red* (*dark*) color edges indicate direct interaction (neighbors), *orange* (*mild*) color edges indicate a tie among developers to whom a release team member has direct interaction, and *grey* (*pale*) color represents the remaining participants. The biggest node close to the center is the release team mailing list

and the proportion of discussions related to release management, we can infer that *desktop-devel-list* is relevant for the release management process. The list is described as being for general use, however the topics that emerged and the influence of the *release-team* indicate that this mailing list is used for release management communication and coordination within the ecosystem.

The release team members tend to participate in a wide range of discussions. In the social network, this would mean being connected to the majority of the nodes. However, a release team member is less prone to participate in a discussion where other release team members are already participating. According to our interviewees, if a release team member is already taking care of a discussion, the other members would leave them to *lead* the discussion. When members of the release team have different opinions, they discuss them in their internal mailing list. Once they reach consensus, one of them goes back to the discussion on the global mailing list and continues the discussion as a representative of the release team.

Other members then do not participate in the public discussion.

> *"When someone from the release team takes ownership of a topic, for instance, in a mailing list, if we agree with this person we can let this person handle the topic and so we do not participate in the discussion, and we also have a lot of discussion on the release team mailing list and on IRC, which does not appear on desktop-devel-list."*

Members of the release team also contribute to other areas of the project, which are not restricted to software programming. As a matter of fact, some release team members do not code. This provides the release team with awareness and a closer connection to different areas of the project. The areas can be documentation, translation, accessibility, quality assurance, system administration, and, in general, anything necessary to develop a software project. Most of these areas are represented by teams. The more diverse the release team members are, the most representative the release team would be

of the whole project. That keeps the release team aware of the different points of view of contributors earlier in the development process; and by increasing the areas of participation, the teams would be more willing to listen the release team. In other words, the more diverse the release team members are, the wider its coverage would be. When a member leaves, the release team members discuss which areas need improvement and recruit someone accordingly.

"When somebody wants to leave the release team, we do discuss which areas we would like to have more coverage [by release team members] ... to get a better flow of information and to also have some influence in both directions."

The diversity in the composition of the release team works in both ways: it helps the release team to have first hand insights from a variety of teams or projects, and it helps projects and teams to have first hand access to the release team. Furthermore, the diversity helps the release team to better address different sensibilities within the ecosystem, especially in controversial topics.

"Is is little bit about peer groups, [...] who works with whom [...] in a topic that is a bit heated [...] somebody from the release team who is in the same team that this person answers, or I even ask 'you have a better connection to this person, could you provide the answer?' "

The release team is comprised of nine individuals, and to support diversity across supporting organizations, the release team does not allow that more than two members share the same affiliation—directly or indirectly [74].

> *In GNOME, the release team members participate in most of the discussions, although rarely more than one member participate in the same discussion. This unwritten policy reduces the possibility of showing conflicting views between each other in public forums, minimizing any potential confusion among developers. Through social network analysis we visualized such behaviour, which is inline with the diversity in the composition of the release team.*

4.4 What are the release management tasks in a FOSS ecosystem?

The objectives of the release team are (1) defining the requirements of GNOME releases, (2) coordinating and communicating with projects and teams; and (3) shipping a release within defined quality and time specifications. The release team makes all the decisions regarding release management and is accountable to the Board of Directors. While the Board of Directors is not directly involved in day-to-day activities, it has the power to dissolve the release team. The GNOME Foundation members are encouraged to participate in the discussion to affect the decisions of the release team. In extreme cases of disagreement, members can propose a referendum [74].

As we described in Section 4.2, the release schedule of GNOME guides the type and timing of coordination activities discussed in the main communication channel. Each of the resulting categories of communication can be mapped to the release team objectives. Thus, the first objective maps to *Request for comments* and *Proposals and discussions*; the second maps to *Proposals and discussions*, *Schedule reminders*, and *Request for approval*; and the third objective maps to *Announcement*, *Schedule reminders*, and *Request for approval*.

From our analysis of mailing lists, the release team leads the discussions to define the requirements of a GNOME Release. These discussions occur during the *feature proposals* stage at the beginning of every release cycle. GNOME releases follow a time-based release cycle of six months (26 weeks). Figure 2 illustrates the release cycle. A release cycle starts immediately after a major release. The stable version is kept in a branch for bug fixes, whereas the regular development continues in the main branch. The release cycle starts with a release meeting to evaluate the last cycle, and to discuss the next cycle [76]. At this point, the release team creates a schedule of activities to deliver a release on time. The release team keeps the teams informed of this schedule, verifying that the activities are completed as required. In one of the interviews, a release team member described, in general terms, how the schedule is created:

"GNOME is expected [by distributions] to release a new version at the end of March and at the end of September. This is well known among most developers. [However,] we propose a schedule trying, for example, to not have a release on Christmas or Easter weekend, or when people are expected to travel, like before or after GUADEC [the GNOME conference]."

During the first 20 weeks of a release cycle, there are development releases every four weeks [77]. In the 21st week, a stabilization phase begins with the release of a first beta version. The stabilization is characterized by an incremental *freeze* period: after this point, every change requires the approval of the release team. When the first *freeze* starts, it is not allowed to introduce new features nor user interfaces changes; the former is to focus in fixing issues and finishing the features already started, and the later, is to give enough time to the Documentation Team to make the manuals. When the second *freeze* starts,

Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 16 of 24

a new restriction is added on top of the previous ones: it is forbidden to change any strings exposed to the users; this allows the Documentation Team to polish the screen shots for the documentation, and the Translator Team to polish the translation of the user interface of applications to multiple languages. The third and last *freeze* focuses on testing and fixing critical bugs, no changes can be made without approval by the release team. In week 26, the final version is released and a new cycle begins.

The schedule planning takes in account activities such as short-term and long-term proposals and discussions, development, testing, and translations. As we reported in Section 4.2, the release schedule drives the main types of communication for coordinating the ecosystem. Based on the schedule, developers propose new features early in the release cycle which are openly discussed in the mailing list. The proposals are expected to have a working plan and the approval of the maintainers in charge of the projects involved. To help reaching consensus, the release team may guide the discussions to increase the community participation, make sure that all major concerns from developers are raised and addressed, minimize potential conflicts between proposals and long term plan, and keep under control new dependencies. Because of the lack power over developers, the release team must rely on their social skills and technical merits before taking decisions that may be controversial within the ecosystem (see Section 4.5.1); however, the diversity of backgrounds among release team members is useful to understand and better address the variety of projects and teams within the ecosystem.

Our data shows how, to ship a release within defined quality and time specifications, the release team takes control of the changes planned to be included in the release. As the release date approaches, the project maintainers require approval to make changes in their projects. The release team also coordinates the release notes, working with developers and teams—such as the marketing team—to write cohesive release notes for GNOME.

To make a release, the release team builds every component and validates that the software runs as expected. If a component fails to build the release team will get in touch with the developers of the failing component to fix the build. Release team members acknowledged that this is one of the most time-consuming and challenging tasks. The purpose of continuously building and testing a planned release is to enable the release team to monitor the quality of the product during the whole release cycle, determine critical bugs and follow-up with developers to fix them. They also need to coordinate with distributors of GNOME regarding potential issues.

Our interviews confirm that within the release team there are no official roles. The tasks are self-assigned among release team members themselves, who volunteer to perform a task depending of their experience, workload, or interest.

> The release team defines what a GNOME release is, sets the schedule, coordinates with projects and cross-cutting teams to reach the goal on time, integrates and validates the product as a whole, and releases GNOME. Because the release team lacks power over the developers, it relies on building consensus on technical merits to convince the developers of their judgment.

4.5 What are the challenges that release managers face in a FOSS ecosystem?

From our analysis and interviews, we identified the four major challenges that release managers face in the GNOME ecosystem, they: (1) need to coordinate projects and teams of volunteers without direct power over them (2) keep the build process manageable (3) monitor for unplanned changes, and (4) test the GNOME release.

4.5.1 *Coordinate projects and teams of volunteers without direct power over them*

GNOME contributors participate as volunteers, even though some of them are paid by external companies for their involvement. Projects are "*owned*" by the contributors who actively work on them, and these people make decisions regarding their projects.

"Maintainers have the last word most of the time. If the conflict is about a maintainer not agreeing with your vision ..., with specific technical decision, then it is [the maintainer's call]."

The release team lacks of power to mandate developers to perform a task, it relies on building consensus based on technical merit and engaging developers to embrace the release process as theirs. One challenge the release team faces is to convince developers of its judgment and knowledge in the release process.

"It is difficult to coordinate well; there are so many people, so many teams. You need to be sure that everybody is aware on what is going on, that everybody is really involved when you need input. It is hard to coordinate people, it is really hard ... we try to do the best we can, but still is not perfect."

The release team builds awareness of the whole release process by increasing the community participation. The time-based schedule facilitates this task by providing the same information to everyone beforehand [53], providing developers a sense of ownership of specific tasks and to become more involved in the process. This emphasizes

Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 17 of 24

the importance of social skills and power of persuasion of the release team members.

4.5.2 Keep the build process manageable

GNOME is composed of multiple piece of software, each one with its own set of dependencies. When the number dependencies grows, building the whole GNOME becomes cumbersome as it takes longer, and with more points of build failures. As a consequence, less volunteers build and test the whole GNOME before the release, which in turn increases the workload of the release team.

The release team addresses the scalability issue by keeping the *building stack* as small as possible, however, it is challenging to keep the stack small. We learned this observation directly from the interviews, as a release team member stated:

> "In GNOME 3, we tried to make the stack smaller, [by reducing] the set of modules. For a short while we managed to get it below 200 [dependencies]. But then, new dependencies trap you back and now we have like 220 or so."

Our interviews show that one way to make the *building stack* smaller is by avoid managing external dependencies whenever is possible. To do so, the release team defines two kind of dependencies: system dependencies and regular dependencies. The system dependencies are the preferred external dependencies as they are mature enough to be readily available in the most common distributions. The regular dependencies are any other and must be built by GNOME, they can be software within GNOME or an external dependency.

4.5.3 Monitor for unplanned changes

Changes in the Application Programming Interfaces (API) and Application Binary Interfaces (ABI) of libraries pose a challenge to release managers. The libraries that GNOME provides try to guarantee stability in both API and ABI; thus, any application that uses a public API of a stable series will continue working with future releases without recompilation. Because the GNOME stack has several libraries, each one maintained by different people, it is challenging to track unintentional breakages before a release. Some changes in the API or ABI might work well in some configurations, but break in others; or may be specific for a platform or architecture. To illustrate this observation, a release team member indicated:

> "A change ... that works fine in my local system, maybe breaks some application somewhere else in the stack, or maybe it breaks only on a 32-bits system that I don't test locally because my laptop is 64-bits. Or in some parts of our stack ... we have to be worried about Windows or FreeBSD."

To enable applications to be ported to multiple operating systems and architectures, in the GNOME ecosystem the core libraries are multi-platform. However, the support for other operating systems is limited by the number developers working on them. Because the main platform of development is Linux, some breaks in other operating systems (Windows or FreeBSD) may be noticed late in the development process if nobody monitors them.

Each project can decide on its own whether to add a new public API. However, the release team monitors the API and ABI stability, and makes sure the API documentation is up-to-date. To this end, the release team needs to detect API changes and make sure they follow the programming guidelines.

4.5.4 Test the GNOME release

The number of projects to coordinate, as well as dependencies on external projects, make cumbersome testing the latest development version of GNOME. These quality assurance activities are performed by a small group of developers, mainly the release team as who is in charge of continuous integration. In the words of a release team member, continuous integration is a necessity:

> "[full automated continuous integration] would allow us [to be] more aggressive: if something causes a problem, we can just back it out. Nowadays we commit something [that] works in our systems, and people keep working on top. [Months] later, we find out ... problems somewhere else, but nobody noticed them because nobody managed to build the whole tree and actually test it."

OSTree [79] is a project that aims to address this issue by continuously building GNOME and providing a testable system ready to be downloaded and run. The release team uses it to build and test GNOME. Although promising, it may require intervention from the release team when some builds fail.

> *The challenges of the release team in GNOME are associated with the size and complexity of managing multiple independent projects, and developed by volunteers in a distributed setting.*

5 Discussion

In this section, we discuss our findings and present the lessons we learned from studying release management in the GNOME ecosystem. Our empirical investigation was based on three theories: the media richness theory, the

Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 18 of 24

channel expansion theory, and the theory of shared understanding. Based on these, we arrived to a set of lessons learned.

5.1 Media richness theory

Media richness theory [14] argues that the communication channels and organizational structure play an important role to reduce uncertainty and ambiguity in the organizational processes. In a community of developers, the selection of a communication channel delimits who can participate and how much can be communicated in a period of time. In FOSS, developers need to find a balance between inclusion (number of participants) and the "bandwidth" provided by the communication channel. The "bandwidth" in a communication channel corresponds to the number of cues in which an information can be transferred [14]. The cues can be verbal (speech, writing) or non–verbal (seeing, touching, tone of voice, vocal inflection, physical gesture, smelling, touching) [14, 16, 17, 62].

In our observations, participants value synchronous and asynchronous communication. Therefore, an ecosystem may consider using a combination of both type of media channels: asynchronous and synchronous. Asynchronous channels, like mailing lists, allow a wide range of participants, can be an egalitarian medium (explained in detail below), and be archived (and searchable). Synchronous channels, like face–to–face interactions and video-conferencing, allow higher bandwidth, but are less "open" as it may exclude participants as they cannot attend to a conference, or some participants may not be as fluent as English-native speakers.

Lesson 1: Ensure that the release team follows the main communication channels used by developers. The release team needs to communicate in a variety of ways. They use electronic channels that vary from asynchronous (such as email and blogs) to more direct, interactive ones (such as IRC). They value face-to-face communication; for this purpose they organize gatherings (such as conferences and hackfests) where the release team can host sessions to address specific issues, or communicate one-on-one with some contributors.

Regardless of having communication channels that provide higher bandwidth, an important factor documented by Olsen and Olsen [56] is the knowledge that the participants have in common, and they are aware that they have it in common; this concept is known as *common ground*. The use of mailing lists can succeed in FOSS

ecosystems because the community of developers have established a common ground. Olson and Olson argue the more common ground people can establish, the easier the communication and the greater the productivity [56]. They also argue that when people have established little common ground, even if they have met in person or using high bandwidth channels, there will be low communication productivity regardless of the communication channel. The properties discussed here are in line with the related research literature on media channels [14, 41], except one that we did not find evidence of being addressed: the difference in English fluency. The asynchronous channels provide a better platform for communication and are more participatory for those who do not speak English fluently. We can argue that is another instance of *"richness media paradox"* [63], where the rich media can simultaneously improve and impair the communication; in this particular case, the rich media can impair the participation of non-native English speakers.

Lesson 2: Provide a common place for coordination for an ecosystem. Every software project has its own communication channel infrastructure, such as their own git repository or mailing list. However, to coordinate multiple projects is necessary a common infrastructure across projects. This facilitates the communication to flow from the release team to the projects and vice versa. In the GNOME ecosystem, this is performed by having an asynchronous ecosystem-wide mailing list for high-level discussions, and an synchronous ecosystem-wide IRC channel for quick feedback. Thus, an asynchronous communication channel (such as mailing list) is ideal for the main coordination activities, and a synchronous channel (particularly face–to–face interactions at conferences or sprints) is good for synchronization of activities among developers.

In a FOSS ecosystem, with the variety of cultural and technical backgrounds of its members, the main communication channel for coordination is asynchronous (usually mailing). Email is egalitarian because it allows contributors with different levels of English skills to participate in equal terms (something that it is hard to achieve in synchronous channels, where contributors with better language skills can dominate a discussion). In an overview of the research literature, we did not find references to language barriers in the use of communication channels in software development, perhaps because they did not study teams with the same level of variability of national origins

as the FOSS ecosystem we studied. This finding may have implication also in any regular FOSS project whose contributors proceed from a variety of cultural and technical backgrounds.

Face—to—face interactions at conferences and hackfests are appreciated by most of the interviewees in spite of its lack of inclusiveness. It is not-inclusive because not all developers have the opportunity to attend to conferences or gatherings (because of limited budget, work restriction, or other restrictions), and even if they attend they might not be present during a particular discussion. In addition, conferences and hackfests provide opportunities for socialization and to establish group identity, both enhance the respect of social norms. These social norms are particularly important to deal with exceptional events, such as, the potential lack of responsiveness of a developer or delays in the development of a given feature [65].

Lesson 3: Consider including both good technical and social skills in a release team. Different visions of the project might create friction between developers and release managers, as their expectations for what should be in a release might differ. Technical skills are needed to understand the technical aspects of the project, and to build consensus on technical merits; technical skills are also needed to gain respect from their peers and to convince developers of their judgment. The release managers need social skills to convince developers to perform the necessary actions to deliver the software on time, especially because release managers lack direct power over developers.

Achieving agreement between participants may be more difficult in electronic media channels than face–to–face interactions [81]. Therefore, virtual teams rely on trust; but it takes time to establish trust in complex environments [32, 43, 50] like in a FOSS ecosystem. Building trust is important for team coordination, as it has been reported that the lack of trust is a barrier to team coordination that geographically distributed organizations face [28, 29].

To build trust, the GNOME release team is composed of members of the community who work on different projects and teams. Therefore, they are exposed to the same issues that other contributors might face, they can raise an issue before it escalates, or they can find better ways to communicate with certain groups of developers. The diversity in the release team composition can provide different backgrounds, which helps enrich the discussions and to increase awareness across different projects and teams.

Lesson 4: Aim for a diverse release team. This implies that its members should be recruited from different teams and with different skill sets. It helps having first-hand knowledge and understanding of the different projects and teams, and to be able to reach everybody in the ecosystem. This diversity is also likely to provide different points of views. They also need to be (or at least have been) members of the teams that they expect to guide. By being "one of them", both sides will be able feel more affinity to the challenges and problems of the other side, especially when the release team makes decisions that contravene the wishes of a given team. In a way, members of the release team are not only release managers, but they are also representatives of the teams. They are expected to make the best decisions that benefit both the ecosystem and the individual teams as a whole.

The GNOME release team has used different strategies to cope with the lack of official power over developers, and the diversity of projects to coordinate. For example, the release team presents an unified position to the ecosystem, informs constantly the ecosystem about the release process, looks pro-actively to reach all members of the ecosystem, and keeps itself aware of the ecosystem. Another strategy followed by the release team is to recruit key contributors from different teams into the release team. The recruited contributors by the release team may or may not be leaders (maintainers) of their corresponding projects, however, they represent different groups of developers.

By comparing the results of the analysis of discussions in the mailing list against the interviews, we found that some release team members underestimate their role because of lack of official power over developers—their decisions could be challenged by developers at any time.

We found that the release team decisions are respected even when there is disagreement. For example, in 2010, the release team decided that GNOME 3.0 was not ready yet. Instead, it would make a maintenance release of the GNOME 2.*x* series (GNOME 2.32). This decision was announced, first, during the major GNOME conference (GUADEC), and second, in the *desktop-devel-list* mailing list [61]. The announcement triggered a strong disagreement by a group of vocal core developers, who argued that this decision would delay: (1) porting applications to the new platform (2) testing the new GNOME Desktop in the wild, where it is actually tested after a release. To highlight their disagreement, some developers hung a banner during GUADEC stating *"Down w/release team" and the symbol of anarchy* (see Fig. 8). In GNOME, the maintainers of each project
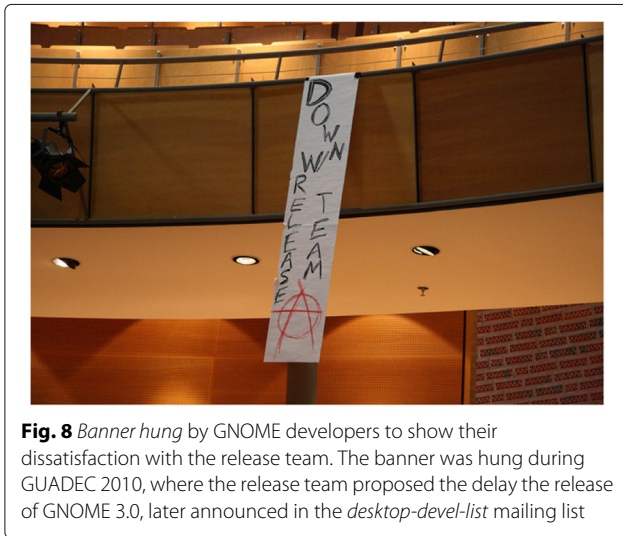
Fig. 8 *Banner hung* by GNOME developers to show their dissatisfaction with the release team. The banner was hung during GUADEC 2010, where the release team proposed the delay the release of GNOME 3.0, later announced in the *desktop-devel-list* mailing list

decide what to release, and therefore, the disagreeing developers could have released newer version of their projects (for GNOME 3.0) regardless of the release team. However, the release team's decision was respected, even though it was unpopular.

> *Lesson 5: Based on lack of power, lobbying and consensus based management must be followed.* To build trust, raise awareness, and keep the ecosystem aligned to the goals of the release process, the release team needs to be surrounded by contributors who are (or can become) "influencers", because these influencers may already be trustworthy in different parts of the ecosystem. Influencers can help the release managers communicate more effectively with the whole ecosystem.

5.2 Channel expansion and shared understanding theories

Channel expansion theory [10] presumes that as individuals gain experience in the use of a channel they are able to use the media channel more effectively; such experience involves the use of the channel itself, the organizational context, the message topic, and the understanding of how their co-participants communicate. Furthermore, according to Dennis and Valacich [16], the development of standards and norms increases with the experience that a group gain, and as a result there may also experience an improvement in the interplay and the tasks they perform over time.

We have seen that the communication flow is flat in GNOME, the communication and coordination is driven

by the release team; every developer participates in the discussions and the maintainers of each project take the decisions. The release team members see themselves as peers of the developers; they build trust based on technical merits to reach consensus and they align the developers towards a common goal: the integrated product.

Michlmayr et al. [50, 52] argue that a set of clear policies and deadlines, which are followed and enforced, helps build trust in complex FOSS projects. Hence, it is important to follow and enforce policies and deadlines, otherwise the effect may be the opposite: the loss of trust in the work of the release managers. As an example of such enforcement, according to Michlmayr [52], release managers can proactively omit or postpone unfinished features.

According to the *theory of shared understanding* [1], the formalization of a process in a software project helps reduce the overall complexity, and once the stakeholders are familiarized with the process itself, they can make assumptions on parts of the organizations that are unfamiliar to them. Paraphrasing Aranda [1], the formalization of the release process in a FOSS ecosystem, enables the coordination of a large group of people, and allows the ecosystem to grow. However, this formalization of processes in our study is constrained to the management of the release process, not how each project develops its own software.

We observed that the formalization of process described by Aranda [1], and Michlmayr et al. [50, 52] correspond to the definition of the release schedule and each one of its stages. After a period of proposals and discussions, the release team decides the features will be part of the official release of GNOME. These features are offered by libraries and applications, which once accepted must follow the release schedule strictly. However, there are also applications that follow the schedule even though they are not required to do so. Again, this dynamic might also be overlooked if projects are studied individually and not in the context of the whole ecosystem.

We found that the communication activity in the *desktop-devel-list* mailing list is tied to the release schedule. Five out of nine discussion categories in a general GNOME development mailing list were in related to release management and showed temporal synchronization with the development cycle schedule. One main reason for this is that some cross-cutting teams increase their participation at specific stages of the process. The documentation and translation teams are more active in the mailing list at the end of the development cycle, reviewing change proposals that might affect their work. The release team members play an active role in the discussion during the whole cycle, but publish reminders at key moments in the development cycle.

Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 21 of 24

According to the discussions we studied in the GNOME ecosystem, the message subjects were predictable and tended to follow a common pattern at every release cycle (RQ_2 in Section 4.2), patterns that were also inline with the objectives of the release team (RQ_4 in Section 4.4). We speculate that this behaviour occur because the GNOME ecosystem is a mature project. We based our speculation in that over time developers find a common ground to use more effectively the communication channels used for coordination [56], which also is inline with the theory of collaboration through open superposition by Howison and Crowston [31].

> *Lesson 6: Help the release team in the coordination process with a well defined schedule.* Once the coordination process is internalized by the community, the release team can focus its efforts on other challenges. In addition, the time–based schedule release provides the release team a powerful tool: even though the release team might not know beforehand the features to be included in any release ahead, it makes it certain when the features need to be discussed, decided and released. The time–based release schedule sets the expectations for developers and stakeholders, enabling them to plan ahead with confidence. The planning helps the ecosystem to develop features incrementally. The tasks that might appear too large are likely to be planned in the long term, and deferred if they cannot be finished on time ("there will be another release in six-months").

A challenge of time-based releases is choosing the right release frequency. Too frequent releases may limit innovation as developers may target features that can be implemented within the release interval. Too far apart releases, may provide long-term stability but also be seen as a sign of stagnation and drive contributors away of the project [51].

Guzzi et al. [26] argue that the development mailing list is one of several communication channels used in FOSS projects. We concur, as we noticed in our case study that developers communicate using IRC, face–to–face interactions at conferences, sprints and hackfests, issue trackers (Bugzilla), among others. Furthermore, Guzzi et al. [26] determined that the mailing list only drives a few of the coordination project discussions (3% among all the discussions in the Lucene project), arguing that as a consequence, the role of mailing lists had changed. For example, both Brooks [9] and Parnas [58] argue that when the tasks are clearly divided, there is less coordination required; and according to Michlmayr [53] a decomposed system has the advantage that allows the division of labour with

limited communication through well defined channels. However, Michlmayr and Fitzgerald [51] argue that regular synchronization helps raise awareness and reduce conflict. In an ecosystem, coordination is important to coordinate multiple projects if they work towards a common integrated product, each project may have its own mailing list, but they still need a common channel for cross–project coordination.

In contrast to Guzzie et al. [26], we state that the use of the mailing lists depends on the project dynamics, and the dynamics of a FOSS ecosystem and regular FOSS projects may differ in terms of complexity and coordination needs. Nevertheless, mailing lists can have multiple uses, and projects and ecosystems may adapt or evolve their use of mailing lists over time. In the long term, developers with an established common ground may use a communication channel to coordinate effectively [56]. We also disclose when and for what to use each channel. For this, we provide the rational from the GNOME release team. We can conclude that release management is behaving different from general development.

> *Lesson 7: Release team work is different from regular software work.* Any software system needs to plan and manage its releases. In large ecosystems of inter-related (but independent) projects this task is much more complex than in a single system. The release team plays a coordination role without participating directly in any particular project. The release management activities are not recorded in commits of any project, but in discussions in various channels, including email, IRC, and even face—to—face. For researchers studying coordination, the release team role can be overlooked if not explicitly controlled for.

6 Conclusions and outlook

We explored the GNOME ecosystem to gain a deeper understanding of its dynamics. We determined the main communication channel used to coordinate the ecosystem, extracted meaningful discussion topics, determined the relevant actors, whose later confirmed and enriched our findings.

The release team is a key player in the communication and coordination among developers in GNOME. The communication coverage that the release team has in the GNOME community is far-reaching. This phenomenon has so far been undocumented. Our interviewees were surprised by this finding, yet they all agreed that it made sense.

In GNOME, the release team members come from a variety of teams or projects, as some of their members acknowledged in the interviews. Some of them are from

the system administrators team, bug squadron, accessibility team, or maintainers of individual projects. This variety allows the release team to monitor and address almost all communications. Our interaction analysis could be beneficial for the release team, either to detect communication anomalies in time or to discard irrelevant issues faster.

The release team leads the coordination efforts in the GNOME ecosystem, it is the glue that keeps multiple projects and teams working together towards a goal. It is a crucial team for the success of GNOME, even if some of its members write little or no code at all.

The focus of our case study was an ecosystem "in-the-small". However, projects in an ecosystem are not developed in a vacuum, they have dependencies and other ecosystem may depend on them, as well. A study of an ecosystem "in-the-large" may show interactions between software ecosystems, with individuals acting as brokers between them. As our results show, GNOME projects already rely on external libraries, utilities, systems, and organizations that at least the release team needs to coordinate with. Some of those external dependencies may be part of another ecosystem, with developers in common ecosystems who may act as bridges between ecosystems.

The operational details of release management among ecosystems might vary. The characteristics of the release team, tasks, challenges, the interaction with projects and teams, and the lessons learned of this case study can be compared against other ecosystems in future research. Specially, how the release management in ecosystems cope with similar challenges than the GNOME ecosystem. The similarities may help build a theory of coordination and communication for release management in FOSS ecosystems.

Abbreviations
ABI: Application binary interfaces; API: Application programming interfaces; FOSS: Free and Open source software; IRC: Internet relay chat; LDA: Latent dirichlet allocation; MSR: Mining software repositories

Authors' contributions
GPC carried out data-collection and analysis, conducted the interviews, and had a major role in drafting the manuscript. EK and LS participated in the discussion of the analysis, and drafted the manuscript. DMG provided guidance with the research design, interviews, analysis, and drafted the manuscript. All authors read and approved the final manuscript.

Author details
[1]Department of Computer Science, University of Victoria, 3800 Finnerty Road, Victoria, British Columbia, Canada. [2]Chalmers | University of Gothenburg, Gothenburg, Sweden.

References
1. Aranda J. A Theory of Shared Understanding for Software Organizations: PhD thesis, University of Toronto; 2010.
2. Bachmann A, Bernstein A. When process data quality affects the number of bugs: Correlations in software engineering datasets. In: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). Cape Town: IEEE; 2010. p. 62–71.
3. Bastian M, Heymann S, Jacomy M. Gephi: An Open Source Software for Exploring and Manipulating Networks. In: Proceedings of the International Conference on Weblogs and Social Media; 2009. p. 361–2.
4. Berkus J. The 5 Types of Open Source Projects. 2005. Online. Visited on 28 Feb 2013. http://web.archive.org/web/20090130091039/http://powerpostgresql.com/5_types.
5. Bird C. Sociotechnical coordination and collaboration in open source software. In: Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011). Williamsburg: IEEE; 2011. p. 568–73. doi:10.1109/ICSM.2011.6080832.
6. Bird C, Gourley A, Devanbu P, Gertz M, Swaminathan A. Mining Email Social Networks. In: Proceedings of the 3rd. International Workshop on Mining Software Repositories. Shanghai: (MSR 2006); 2006. p. 137–43.
7. Bohn A, Feinerer I, Hornik K, Mair P. Content-Based Social Network Analysis of Mailing Lists. R J. 2011;3(June):11–18.
8. Bosch J. From Software Product Lines to Software Ecosystems. In: Proceedings of the 13th International Software Product Line Conference (SPLC 2009). San Francisco; 2009. p.111–9.
9. Brooks Jr FP. The Mythical Man-Month: Essays on Software Engineering, 1st ed. Massachusetts: Addison-Wesley Publishing Company, Reading; 1975, p. 195.
10. Carlson JR, Zmud RW. Channel Expansion Theory and the Experiential Nature of Media Richness Perceptions. Acad Manag J. 1999;42(2):153–70.
11. Casebolt JR, Krein JL, MacLean AC, Knutson CD, Delorey DP. Author entropy vs. file size in the gnome suite of applications. In: Proceedings of the 6th Ieee International Working Conference on Mining Software Repositories (MSR 2009). Vancouver: IEEE; 2009. p. 91–4.
12. Corbin JM, Strauss A. Grounded theory research: Procedures, canons, and evaluative criteria. Qual Sociol. 1990;13(1):3–21.
13. Creswell JW. Research Design: Qualitative, Quantitative, and Mixed Methods Approaches vol. 2. Thousand Oaks: Sage Publications; 2009, p. 260.
14. Daft RL, Lengel RH. Organization Information Requirements, Media Richness and Structural Design. Manag Sci. 1986;32(5):554–71.
15. de Souza C, Froehlich J, Dourish P. Seeking the Source: Software Source Code as a Social and Technical Artifact. In: Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work (GROUP 2005). Sanibel Island: ACM; 2005. p. 197–206.
16. Dennis AR, Valacich JS. Rethinking media richness: towards a theory of media synchronicity. In: Proceedings of the 32nd Annual Hawaii

Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 23 of 24

International Conference on Systems Sciences (hicss 1999). Maui: IEEE; 1999. p. 1–10. doi:10.1109/HICSS.1999.772701.

17. Dennis AR, Kinney ST, Hung Y-TC. Gender Differences in the Effects of Media Richness. Small Group Res. 1999;30(4):405–37.

18. Easterbrook S, Singer J, Storey MA, Damian D. Selecting Empirical Methods for Software Engineering Research. In: Guide to Advanced Empirical Software Engineering. London: Springer; 2008. p. 285–311.

19. Erenkrantz JR. Release management within open source projects. In: Proceedings of the 3rd Open Source Software … Portland; 2003. p. 51–5.

20. Flyvbjerg B. Five Misunderstandings About Case-Study Research. Qual Inq. 2006;12(2):219–45.

21. Fogel K. Producing Open Source Software: How to Run a Successful Free Software Project. Paramount: O'Reilly Media, Inc.; 2005.

22. German DM. The GNOME Project: A Case Study of Open Source, Global Software Development. Softw Process Improv Pract. 2003;8(4):201–15.

23. German DM, Adams B, Hassan AE. The Evolution of the R Software Ecosystem. In: Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013). Williamsburg: IEEE; 2013. p. 243–52. doi:10.1109/CSMR.2013.33.

24. Goeminne M, Mens T. Towards the Analysis of Evolution OSS Ecosystems. In: Proceedings of the 8th BElgian-NEtherlands Software eVOLution Seminar (BENEVOL 2009); 2009. p. 30–5.

25. Gutwin C, Penner R, Schneider K. Group awareness in distributed software development. In: Proceedings of the Conference on Computer Supported Cooperative Work (CSCW 2004). Chicago; 2004. p. 72–81.

26. Guzzi A, Bacchelli A, Lanza M, Pinzger M, Deursen AV. Communication in Open Source Software Development Mailing Lists. In: Proceedings of the 10th Working Conference on Mining Software Repositories (MSR 2013). San Francisco: IEEE; 2013. p. 277–86.

27. Halverson CA, Ellis JB, Danis C, Kellogg WA. Designing task visualizations to support the coordination of work in software development. In: Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW 2006). Banff; 2006. p. 39–48.

28. Herbsleb JD, Grinter RE. Architectures, coordination, and distance: Conway's law and beyond. IEEE Softw. 1999;16(5):63–70.

29. Herbsleb JD, Grinter RE. Splitting the organization and integrating the code: Conway's Law Revisited. In: Proceedings of the 21st International Conference on Software Engineering (ICSE 1999). Los Angeles: ACM Press; 1999. p. 85–95.

30. Herbsleb J, Mockus A, Finholt TA, Grinter RE. An empirical study of global software development: distance and speed. In: Proceedings of the 23rd International Conference on Software Engineering ICSE 2001; 2001. p. 81–90.

31. Howison J, Crowston K. Collaboration through open superposition: A theory of the open source way. MIS Q. 2014;38(1):29–50.

32. Iacono CS, Weisband S. Developing Trust in Virtual Teams. 30th Hawaii International Conference on System Sciences (HICSS) Volume 2: Information Systems Track-Collaboration Systems and Technology. 1997412–420.

33. Jacomy M, Venturini T, Heymann S, Bastian M. ForceAtlas2, a Continuous Graph Layout Algorithm for Handy Network Visualization Designed for the Gephi Software. PLoS ONE. 2014;9(6):98679.

34. Jansen S. Measuring the health of open source software ecosystems: Beyond the scope of project health. Inf Softw Technol. 2014;56(11): 1508–19.

35. Jansen S, Finkelstein A, Brinkkemper S. A sense of community: A research agenda for software ecosystems. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009). Vancouver: IEEE; 2009. p. 187–90.

36. Jergensen C, Sarma A, Wagstrom P. The Onion Patch: Migration in Open Source Ecosystems. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011). Szeged; 2011. p. 70–80.

37. Knauss E, Damian D, Poo-Caamaño G, Cleland-Huang J. Detecting and classifying patterns of requirements clarifications. In: 2012 20th IEEE International Requirements Engineering Conference (RE 2012). Chicago; 2012. p. 251–60. doi:10.1109/RE.2012.6345811.

38. Knauss E, Damian D, Knauss A, Borici A. Openness and requirements: Opportunities and tradeoffs in software ecosystems. In: Proceedings of the IEEE 22nd International Requirements Engineering Conference (RE 2014). Karlskrona, Sweden; 2014. p. 213–22.

39. Koch S, Schneider G. Effort, co-operation and co-ordination in an open source software project: GNOME. Inf Syst J. 2002;12:27–42.

40. Lamkanfi A, Demeyer S, Giger E, Goethals B. Predicting the severity of a reported bug. In: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). Cape Town: IEEE; 2010. p. 1–10.

41. Lanubile F. Collaboration in Distributed Software Development. In: Software Engineering; 2009. p. 174–93.

42. Linstead E, Baldi P. Mining the coherence of GNOME bug reports with statistical topic models. In: 2009 6th IEEE International Working Conference on Mining Software Repositories. Vancouver: IEEE; 2009. p. 99–102.

43. Ljungberg J. Open source movements as a model for organising. Eur J Inf Syst. 2000;9(4):208–16.

44. Luijten B, Visser J, Zaidman A. Assessment of issue handling efficiency. In: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). Cape Town: IEEE; 2010. p. 94–7.

45. Lungu M. Towards reverse engineering software ecosystems. In: Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM 2008). Edmonton; 2008. p. 428–31.

46. Lungu M, Malnati J, Lanza M. Visualizing Gnome with the Small Project Observatory. In: Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR 2009). Vancouver; 2009. p. 103–6. doi:10.1109/MSR.2009.5069487.

47. Lungu M, Lanza M, Gîrba T, Robbes R. The Small Project Observatory: Visualizing Software Ecosystems. Sci Comput Program. 2010;75(4):264–75.

48. Markus ML. The governance of free/open source software projects: monolithic, multidimensional, or configurational? J Manag Governance. 2007;11(2):151–63.

49. Mens T, Goeminne M. Analysing the Evolution of Social Aspects of Open Source Software Ecosystems. In: Proceedings of the 3rd International Workshop on Software Ecosystems (ISWSECO 2011). Brussels; 2011. p. 1–14.

50. Michlmayr M. Quality Improvement in Volunteer Free and Open Source Software Projects Exploring the Impact of Release Management: PhD thesis, University of Cambridge; 2007.

51. Michlmayr M, Fitzgerald B. Time-Based Release Management in Free and Open Source (FOSS) Projects. Int J Open Source Softw Process. 2012;4(1):1–19.

52. Michlmayr M, Fitzgerald B, Stol K-J. Why and How Should Open Source Projects Adopt Time-Based Releases? IEEE Softw. 2015;32(2):55–63.

53. Michlmayr M, Hunt F, Probert D. Release Management in Free Software Projects: Practices and Problems. In: Open Source Development, Adoption and Innovation; 2007. p. 295–300.

54. Mockus A, Fielding RT, Herbsleb JD. A case study of open source software development: the Apache server. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000). Limerick: ACM; 2000. p. 263–72.

55. Mockus A, Fielding RT, Herbsleb JD. Two case studies of open source software development: Apache and Mozilla. ACM Trans Softw Eng Methodol. 2002;11(3):309–46.

56. Olson GM, Olson JS. Distance matters. Human-Computer Interact. 2000;15(2):139–78.

57. Pagano D, Maalej W. How do developers blog? In: Proceeding of the 8th Working Conference on Mining Software Repositories (MSR 2011). Honolulu: ACM Press; 2011. p. 123–32.

58. Parnas DL. On the criteria to be used in decomposing systems into modules. Commun ACM. 1972;15(12):1053–8.

59. Peiwei Mi, Scacchi W. Modeling Articulation Work in Software Engineering Processes. In: Proceedings of the 1st International Conference on the Software Process; 1991. p. 188–201.

60. Perry DE, Porter AA, Votta LG. Empirical Studies of Software Engineering. In: Proceedings of the Conference on The Future of Software Engineering (ICSE 2000); 2000. p. 345–55.

61. Péters F. GNOME 3.0 in March 2011. 2010. Online. Visited on 28 Feb 2014. https://mail.gnome.org/archives/desktop-devel-list/2010-July/msg00133.html.

62. Rasters G. Communication and Collaboration in Virtual Teams Did we get the message? PhD thesis, Radboud Universiteit Nijmegen; 2004.

63. Robert LP, Dennis AR. Paradox of richness: A cognitive model of media choice. IEEE Trans Prof Commun. 2005;48(1):10–21. doi:10.1109/TPC.2003.843292.

Poo-Caamaño *et al. Journal of Internet Services and Applications* (2017) 8:12

Page 24 of 24

64. Robles G, González-Barahona JM, Izquierdo-Cortazar D, Herraiz I. Tools for the Study of the Usual Data Sources found in Libre Software Projects. Intl. J Open Source Softw Process. 2009;1(1):24–45. doi:10.4018/jossp.2009010102.
65. Rocco E. Trust breaks down in electronic contexts but can be repaired by some initial face-to-face contact. In: Proceedings of the Sigchi Conference on Human Factors in Computing Systems - Chi '98. New York: ACM Press; 1998. p. 496–502.
66. Runeson P, Host M, Rainer A, Regnell B. Case Study Research in Software Engineering: Guidelines and Examples. Hoboken: Wiley Blackwell; 2012, p. 256.
67. Sarma A, Maccherone L, Wagstrom P, Herbsleb JD. Tesseract: Interactive visual exploration of socio-technical relationships in software development. In: 2009 IEEE 31st International Conference on Software Engineering. Vancouver: IEEE; 2009. p. 23–33.
68. Scacchi W, Alspaugh TA. Understanding the role of licenses and evolution in open architecture software ecosystems. J Syst Softw. 2012;85(7):1479–94. doi:10.1016/j.jss.2012.03.033.
69. Schackmann H, Lichter H. Evaluating process quality in GNOME based on change request data. In: 2009 6th Ieee International Working Conference on Mining Software Repositories. Vancouver: IEEE; 2009. p. 95–8.
70. Scott JP. Social Network Analysis: A Handbook. Thousand Oaks: SAGE Publications; 2000.
71. Shihab E, Hassan AE. On the use of Internet Relay Chat (IRC) meetings by developers of the GNOME GTK+ project. In: 2009 6th Ieee International Working Conference on Mining Software Repositories (MSR 2009). Vancouver: IEEE; 2009. p. 107–10.
72. The GNOME Accounts Team. The GNOME Project. 2013. Visited on 28 Feb 2014. https://wiki.gnome.org/AccountsTeam/AccountNamePolicy.
73. The GNOME Foundation. GNOME Foundation Bylaws. The GNOME Project. Boston; 2002. Visited on 28 Feb 2014. http://www.gnome.org/wp-content/uploads/2012/02/bylaws.pdf.
74. The GNOME Foundation. GNOME Foundation Charter Draft 0.61. Online. Boston; 2002. Visited on 28 Feb 2014. http://foundation.gnome.org/charter.html.
75. The GNOME Project. GNOME Desktop Development List. 2001. Online. Visited on 28 Feb 2014. https://mail.gnome.org/archives/desktop-devel-list/.
76. The GNOME Release Team. Guide for New Release Team Members. 2011. Online. Visited on 28 Feb 2014. https://wiki.gnome.org/ReleasePlanning/NewReleaseTeamMembers.
77. The GNOME Release Team. Account Name Requirements. The GNOME Project. 2013. Visited on 28 Feb 2014. https://wiki.gnome.org/ReleasePlanning/TimeBased.
78. Vasilescu B, Serebrenik A, Goeminne M, Mens T. On the variation and specialisation of workload - A case study of the GNOME ecosystem community. Empir Softw Eng. 2014;19(4):955–1008.
79. Walters C, Poo-Caamaño G, German DM. The Future of Continuous Integration in GNOME. In: Proceedings of the 1st Intl. Workshop on Release Engineering (RELENG 2013). San Francisco: IEEE; 2013. p. 33–6.
80. Waugh J. Planet GNOME Guidelines. 2003. Online. Visited on 28 Feb 2014. https://wiki.gnome.org/PlanetGnome.
81. Yamauchi Y, Yokozawa M, Shinohara T, Ishida T. Collaboration with Lean Media: How Open-Source Software Succeeds. In: Proceedings of the Conference on Computer Supported Cooperative Work (CSCW 2000). Philadelphia; 2000. p. 329–38.
82. Yin RK. Case Study Research: Design and Methods (Applied Social Research Methods), 4th ed. Thousand Oaks: Sage Publications; 2008.