Release Management in Free and Open Source Software Ecosystems

by

Germán Poo-Caamaño
B.Sc., Universidad del Bío-Bío, 1994
M.Sc., Universidad de Concepción, 2010

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

© Germán Poo-Caamaño, 2016
University of Victoria

This dissertation is licensed under a Creative Commons Attribution-ShareAlike 4.0
International License.
Release Management in Free and Open Source Software Ecosystems

by

Germán Poo-Caamaño
B.Sc., Universidad del Bío-Bío, 1994
M.Sc., Universidad de Concepción, 2010

Supervisory Committee

Dr. Daniel M. German, Supervisor
(Department of Computer Science)

Dr. Hausi A. Müller, Departmental Member
(Department of Computer Science)

Dr. Issa Traoré, Outside Member
(Department of Electrical and Computer Engineering)
Releasing software is challenging. To decide when to release software, developers may consider a deadline, a set of features or quality attributes. Yet, there are many stories of software that is not released on time. In large-scale software development, release management 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 for releasing software according to a schedule.

In this work, we investigate the release management process in two large-scale FOSS development projects. In particular, our focus is the communication in the whole release management process in each ecosystem across multiple releases. The main research questions addressed in this dissertation are: (1) How do developers in these FOSS ecosystems communicate and coordinate to build and release a common product based on different projects? (2) What are the release management tasks in a FOSS ecosystem? and (3) What are the challenges that release managers face in a FOSS ecosystem?

To understand this process and its challenges better, we used a multiple case study methodology, and collected evidence from a combination of the following sources: documents, archival records, interviews, direct observation, participant observation, and physical artifacts. We conducted the case studies on two FLOSS software ecosystems: GNOME and OpenStack. We analyzed over two and half years of communication in each ecosystem.
and studied developers’ interactions. GNOME is a collection of libraries, system services, and end-user applications; together, these projects provide a unified desktop—the GNOME desktop. OpenStack is a collection of software tools for building and managing cloud computing platforms for public and private clouds. We catalogued communication channels, categorized coordination activities in one channel, and triangulated our results by interviewing key developers identified through social network analysis.

We found factors that impact the release process in a software ecosystem, including a release schedule positively, influence instead of direct control, and diversity. The release schedule drives most of the communication within an ecosystem. To achieve a concerted release, a Release Team helps developers reach technical consensus through influence rather than direct control. The diverse composition of the Release Team might increase its reach and influence in the ecosystem. Our results can help organizations build better large-scale teams and show that software engineering research focused on individual projects might miss important parts of the picture.

The contributions of this dissertation are: (1) an empirical study of release management in two FOSS ecosystems (2) a set of lessons learned from the case studies, and (3) a theory of release management in FOSS ecosystems. We summarize our theory that explains our understanding of release management in FOSS ecosystems as three statements: (1) the size and complexity of the integrated product is constrained by the release managers’ capacity, (2) release management should be capable of reaching the whole ecosystem, and (3) the release managers need social and technical skills. The dissertation discusses this theory in the light of the case studies, other research efforts, and its implications.
Contents

Supervisory Committee ii
Abstract iii
Table of Contents v
List of Tables ix
List of Figures x
Acknowledgements xii

1 Introduction 1
  1.1 Research Statement and Scope 3
  1.2 Overall Methodology 3
  1.3 Contributions 6
  1.4 Structure of the Dissertation 7

2 Background 9
  2.1 Software Ecosystems 9
    2.1.1 Definition of Software Ecosystem 10
    2.1.2 Research Scope of Software Ecosystems 10
  2.2 Release Management 12
    2.2.1 Release Management Strategies 12
    2.2.2 Release Management as an Optimization Problem 14
  2.3 Social Aspects in Software Development 15
    2.3.1 Communication and Communication Channels 15
    2.3.2 Communication on Mailing Lists 17
    2.3.3 FOSS Governance 19
    2.3.4 Social Network Analysis in FOSS Projects and Ecosystems 21
2.4 The GNOME Project .................................................. 23
   2.4.1 Cross-cutting Teams ........................................ 25
   2.4.2 Relation Between Projects and Cross-cutting Teams .... 25
2.5 The OpenStack Project .............................................. 26
2.6 Summary .............................................................. 28

3 Case Study: The GNOME Ecosystem ................................. 30
   3.1 Study Design ....................................................... 31
      3.1.1 Communication Channel Selection ........................ 31
      3.1.2 Data Collection and Cleaning .............................. 32
      3.1.3 Analysis ....................................................... 32
      3.1.4 Social Network Analysis .................................. 35
      3.1.5 Interviews ..................................................... 36
   3.2 Background: The Release Process in GNOME ................. 38
      3.2.1 Release Management and the Release Team ............. 38
      3.2.2 The Release Cycle .......................................... 38
   3.3 Findings ........................................................... 39
      3.3.1 How do developers in the GNOME ecosystem communicate and coordinate to build and release a common product based on different projects? ............................................. 40
      3.3.2 What are the release management tasks in the GNOME ecosystem? 58
      3.3.3 What are the challenges that release managers face in the GNOME ecosystem? ............................................. 60
   3.4 Discussion .......................................................... 64
      3.4.1 Lessons Learned .............................................. 65
   3.5 Threats to Validity .................................................. 67
      3.5.1 Construct Validity ......................................... 67
      3.5.2 Internal Validity ............................................ 68
      3.5.3 External Validity .......................................... 68
      3.5.4 Reliability .................................................. 68
   3.6 Summary ............................................................ 68

4 Case Study: The OpenStack Ecosystem .............................. 71
   4.1 OpenStack ........................................................ 72
   4.2 Study Design ....................................................... 73
4.2.1 Communication Channel Selection ........................................... 73
4.2.2 Data Collection and Filtering ............................................... 74
4.2.3 Analysis ................................................................................. 77
4.2.4 Social Network Analysis ....................................................... 77
4.2.5 Interviews ............................................................................. 79

4.3 Background: The Release Process in OpenStack .......................... 80
4.3.1 Release Management, Release Team and Project Team Leaders .... 80
4.3.2 The Release Cycle ................................................................. 80

4.4 Findings ................................................................................... 82
4.4.1 How do developers in the OpenStack ecosystem communicate and coordinate to build and release a common product based on different projects? ................................................................. 82
4.4.2 What are the release management tasks in the OpenStack ecosystem? ................................................................. 101
4.4.3 What are the challenges that release managers face in the OpenStack ecosystem? ................................................................. 103

4.5 Discussion ............................................................................... 107
4.5.1 Lessons Learned ................................................................. 110

4.6 Threats to Validity ................................................................... 111
4.6.1 Construct Validity ................................................................. 112
4.6.2 Internal Validity ................................................................. 112
4.6.3 External Validity ................................................................. 113
4.6.4 Reliability ................................................................. 113

4.7 Summary .................................................................................. 113

5 Discussion and Theory ................................................................. 116
5.1 Building an Empirical Theory ...................................................... 117
5.2 Theory of Release Management in FOSS Ecosystems ..................... 119
5.2.1 The Size and Complexity of the Ecosystem’s Integrated Product is Constrained by the Release Managers Capacity ................................. 120
5.2.2 Release Management Should Reach the Whole Ecosystem to Increase Awareness and Participation ...................................................... 121
5.2.3 The Release Managers Need Social and Technical Skills ............. 124

5.3 Discussion and Implications of the Theory .................................... 124
5.3.1 Evolution in the Use of a Communication Channels for Release Management ...................................................... 125
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.2 Communication Channels Used for Coordination</td>
<td>127</td>
</tr>
<tr>
<td>5.3.3 Governance and Release Management</td>
<td>129</td>
</tr>
<tr>
<td>5.3.4 Organizational Structure and Communication Flow</td>
<td>131</td>
</tr>
<tr>
<td>5.3.5 (Re)definition of the Product Delivered by the Ecosystem</td>
<td>132</td>
</tr>
<tr>
<td>5.3.6 Management of Dependencies</td>
<td>134</td>
</tr>
<tr>
<td>5.3.7 Continuous Integration and Testing</td>
<td>136</td>
</tr>
<tr>
<td>5.4 Summary</td>
<td>138</td>
</tr>
<tr>
<td>6 Conclusions and Future Work</td>
<td>140</td>
</tr>
<tr>
<td>6.1 Contributions</td>
<td>141</td>
</tr>
<tr>
<td>6.1.1 Lessons Learned</td>
<td>142</td>
</tr>
<tr>
<td>6.1.2 A Theory of Release Management in FOSS Ecosystem</td>
<td>143</td>
</tr>
<tr>
<td>6.2 Future Work</td>
<td>143</td>
</tr>
<tr>
<td>Bibliography</td>
<td>145</td>
</tr>
<tr>
<td>A Human Research Ethics Board Approval</td>
<td>163</td>
</tr>
<tr>
<td>B Interviews</td>
<td>164</td>
</tr>
<tr>
<td>B.1 In Person Interviews</td>
<td>164</td>
</tr>
<tr>
<td>B.1.1 General questions</td>
<td>164</td>
</tr>
<tr>
<td>B.1.2 Study-related questions</td>
<td>166</td>
</tr>
<tr>
<td>B.2 Via email Interviews</td>
<td>169</td>
</tr>
<tr>
<td>B.2.1 Use of communication channels</td>
<td>170</td>
</tr>
<tr>
<td>B.2.2 Roles</td>
<td>171</td>
</tr>
<tr>
<td>B.2.3 Conflict resolution</td>
<td>171</td>
</tr>
<tr>
<td>C Application for Qualitative Analysis</td>
<td>174</td>
</tr>
<tr>
<td>D Contributions</td>
<td>176</td>
</tr>
</tbody>
</table>
List of Tables

Table 2.1 Types of projects in the OpenStack ecosystem. 27
Table 3.1 Summary of communication channels found in the GNOME ecosystem. 41
Table 3.2 Summary of themes found, grouped in related and unrelated to release management activities. 47
Table 3.3 Summary of discussions and messages per category, grouped in related and unrelated to release management activities, and sorted alphabetically. 54
Table 4.1 Summary of communication channels found in the OpenStack ecosystem. 84
Table 4.2 Summary of categories found, grouped in related and unrelated to release management activities. 91
Table 4.3 Summary of discussions and messages per theme, grouped in related and unrelated to release management activities, and sorted alphabetically. 98
Table 5.1 Key concepts for release managers according to our theory of release management in FOSS ecosystems. 125
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Release Management Strategies</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Excerpt of an e-mail that shows the annotations performed by the researcher (in red). First, we looked the subject, then the content of the discussion whenever was necessary. We added a label based on the main topic of discussion.</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>The same idea than in the first step, but summarized and aggregated in one single spreadsheet. It is shown the subject, the theme, number of messages, and duration of the discussions.</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>GNOME six-month release schedule in weeks and its milestones.</td>
<td>40</td>
</tr>
<tr>
<td>3.4</td>
<td>Request for comments during the release cycle.</td>
<td>48</td>
</tr>
<tr>
<td>3.5</td>
<td>Proposals and discussions during the release cycle.</td>
<td>49</td>
</tr>
<tr>
<td>3.6</td>
<td>Announcements during the release cycle.</td>
<td>50</td>
</tr>
<tr>
<td>3.7</td>
<td>Schedule reminders during the release cycle.</td>
<td>51</td>
</tr>
<tr>
<td>3.8</td>
<td>Request for approvals during the release cycle.</td>
<td>53</td>
</tr>
<tr>
<td>3.9</td>
<td>Scope of interaction of Release Team members.</td>
<td>56</td>
</tr>
<tr>
<td>3.10</td>
<td>GNOME six-month release schedule in weeks and its milestones.</td>
<td>59</td>
</tr>
<tr>
<td>4.1</td>
<td>Excerpt of an e-mail that shows an example on how we identified the topic specific to a given project.</td>
<td>75</td>
</tr>
<tr>
<td>4.2</td>
<td>Excerpt of an e-mail that shows the annotations performed by this researcher. First, we looked the subject, then the content of the discussion whenever was necessary. We added a label based on the main topic of discussion. The excerpt also shows an example on how we identified the topic specific to a given project.</td>
<td>78</td>
</tr>
<tr>
<td>4.3</td>
<td>OpenStack six-month release schedule in weeks and its milestones.</td>
<td>82</td>
</tr>
<tr>
<td>4.4</td>
<td>Request for comments during the release cycle.</td>
<td>92</td>
</tr>
<tr>
<td>4.5</td>
<td>Proposals and discussions during the release cycle.</td>
<td>93</td>
</tr>
<tr>
<td>4.6</td>
<td>Announcements during the release cycle.</td>
<td>95</td>
</tr>
</tbody>
</table>
Figure 4.7  Reminders during the release cycle. .......................... 96
Figure 4.8  Request for decision during the release cycle. .............. 97
Figure 4.9  Scope of interaction of Release Team members. ......... 100
Figure 4.10 OpenStack six-month release schedule in weeks and its milestones. 101

Figure C.1  Application used for coding and abstracting themes from the mailing list discussions. .......................... 175
Acknowledgements

Over the past six years I received support and encouragement from a great number of individuals. I owe my sincere and earnest gratitude to my advisor, Daniel M. German, for the support, guidance and patience he showed towards me throughout my dissertation writing.

I would like to thank my supervisory committee of Hausi A. Müller and Issa Traoré. Audris Mockus graciously agreed to be my external examiner, and provided me with invaluable feedback. I would also like to thank the contributors of GNOME and OpenStack, who took part in my case studies, for generously sharing their time, ideas, and feedback.

Along the way, I have also had helpful conversations with Valeria Cortés, Veronika Irvine, Adrian Schröter, Eric Knauss, Irwin Kwan, Indira Nurdiani, Carlos Gómez, and Peter Rigby. Leif Singer showed me how to approach academic writing, and pair-writing with him was very valuable. I am very grateful to Jorge Aranda for the insightful discussions we had on numerous topics. I have learned much through our conversations, and many times they helped me to clear my mind. I am obliged to Lorena Castañeda and Eirini Kalliamvakou who supported me and who played an important role as my gatekeepers to Hausi and Daniel, respectively. I would also like to show my gratitude to the staff, Erin, Jen, Nancy, Wendy, Brian, Paul, and Tom, who were cheerful, gentle and ready to help me.

Outside of the Computer Science department, plenty of people kept me sane and happy. The Griffin family, especially Su and Jim for adopting me during the last year of my dissertation, being very kind hosts, and helping me improve my poor English. I truly appreciate their friendliness and support, which were fundamental to keep me focused in the last mile of my journey. I am very grateful for the friendly staff at Habit Coffee, especially Courtney, Madison, and Shannon for their kindness and welcoming attitude towards me. It is here where I wrote most of this dissertation; I also learned about coffee, writers, local indie music, and had the chance to discuss research and science with people from backgrounds different than mine. The Victoria Go Club kept me intellectually challenged, and through this ancient game I gained more interest in deep learning. Seymour, Dave, and Joshua
encouraged me to approach the Go board from multiple angles, continuously making ques-
tions to keep me focused on the big picture; this is something I can relate to the evolution
of my research work.

I cannot highlight enough the importance of Victoria, and the people I met here, on the
success of my dissertation. I am very grateful for the kindness of its inhabitants, and the
beauty of the city which help me stay sane through the years.

Although geographically distant, I cannot thank my family enough for instilling in me
a random assortment of values and work ethics that contributed to this dissertation, and for
reminding me of my dreams as a child, even though back then I did not know what they
actually meant. Furthermore, my uncle Ricardo will always be a source of inspiration for
me; he will forever be one of my role models.

Finally, my deepest gratitude goes to my beautiful wife, Tatiana, for being a living
example of perseverance, and for giving me a push to finish my dissertation... for believing
in me more than I ever would. My companion, my beacon, my informal advisor, and the
mother of my source of joy: Sebastián.
To Chiquitito.
Chapter 1
Introduction

“The proper place to study elephants is in the jungle, not the zoo.”

—Ephraim R. McLean on emerging theories from empirical research

An important problem in software development is to decide when the software should be released. The decision might be influenced by a deadline, a set of features or quality attributes. It is not uncommon to hear stories of software that is not released on time [18, 76, 167, 166]. Releasing software is challenging.

The software industry faces a challenge to decide the right time of releasing a software product: only a released software helps obtain benefits for the organization but the software must be complete enough to be useful to the users. During the development, the software feeds expectations and consume organizational resources such as time, person month, and power energy. Once the software is released it serves to a purpose for both users and the organization that develops it. Additionally, the release time might have an impact in the adoption of the software, especially if the release can be done ahead of the competency.

Although the software development processes might differ between Free and Open Source Software (FOSS) projects and industry, the former also faces challenges. Wider adoption of the software can increase the possibilities to attract interest from the public, receive funds for their activities, grow the developers base, and consequently, alleviate the maintenance duties of its developers. FOSS projects have the challenge to keep the developers motivated and aligned towards a common goal. At the end of the day, FOSS projects must coordinate a distributed team of volunteers in order to align their work for a release [113].

Given the difficulties of following a release schedule, many FOSS ecosystems start with an “open schedule”, where releases are made at what appears to be random intervals, when the developers decide that the software is ready to be released. Over time, some
ecosystems have evolved to time-based releases where projects define in advance a detailed schedule, from a period to decide the features to include until the date the software will be released [109]. Projects that follow time-based releases prioritize the schedule over features, if the implementation of a feature is not ready on time, then the software is released without it. The result is a predictable process that benefits anybody interested in the project, from users to integrators.

If releasing a single software is challenging, then consider the challenges to release a product composed of a series of software pieces integrated cohesively like a whole. Each of these pieces developed independently of each other, with distributed teams of developers, different motivations, many of them working as volunteers. And yet, many of these are able to produce software and release it on time. The developers of each of these projects must communicate and coordinate effectively to achieve the goal of releasing a cohesive product.

A FOSS ecosystem is a set of independent, interrelated FOSS applications that operate together to deliver a common user experience. Examples of these ecosystems are the Linux distributions (such as Debian), KDE and GNOME (GUI set of applications for the desktop), the R ecosystem (R language, libraries and tools that work together). As such, the release management in a FOSS ecosystem would be significantly more difficult that the management of the release of any of its applications alone. Release managers of an ecosystem need to coordinate the goals and schedules of multiple teams to be able to deliver, from the point of view of the user, one single release.

FOSS ecosystems are complex organizational structures [14, 19, 103] and present unique challenges, such as (1) lack of hierarchical structure, usually without power to tell what contributors should work on (2) need to coordinate contributors geographically located in different parts of the globe, and (3) reach consensus over multiple interests of the contributors [29]. Yet, in these ecosystems, developers work together, reach consensus, and are able to deliver complex products [108].

Ecosystems are becoming common [74, 98]. Release management is challenging, and some ecosystems do it effectively [108, 110]. It is important to understand how ecosystems manage releases because learning from successful ones might help others. However, release management is an area that has received relatively little attention by the software engineering researchers [166]. However, to our knowledge, the requirements and challenges that release managers face in software ecosystems have not been explored.
1.1 Research Statement and Scope

The research goals of this dissertation are as follows: (1) to understand through an empirical study the release management in a FOSS software ecosystem, and (2) based on the empirical study extract lessons and issue recommendations to release managers of FOSS ecosystems and those building tools for it.

For that purpose, we empirically study two large ecosystems, on which we conduct case studies on how these projects do release management. We examine two high profile ecosystems: GNOME and OpenStack. Both ecosystems are large, and have a history of release management that enables them to deliver a new version of an integrated product every six months [108, 110, 135]

We studied the communication of the whole release management process in each ecosystem across multiple releases focusing on the following research questions:

Q1. In the context of release management, how do developers in these FOSS ecosystems communicate and coordinate to build and release a common product based on different projects?

Q1.1. What are the communication channels used for release management?
Q1.2. How do developers communicate and coordinate for release management?
Q1.3. Who are the key actors in the release management process?

Q2. What are the release management tasks in a FOSS ecosystem?

Q3. What are the challenges that release managers face in a FOSS ecosystem?

Based on our findings, we extracted lessons and recommendations on how to approach the release management process in FOSS ecosystems.

1.2 Overall Methodology

To accomplish the research goal, we used a multiple case study methodology as described by Easterbrook et al. [35]. The evidence collected through case study research may come from any combination of the following sources: documents, archival records, interviews, direct observation, participant observation, and physical artifacts [171, 172]. We performed each study using multiple data sources. We relied on documents (web pages, wiki pages, personal journals or blog posts), archival records (mailing lists), direct observation (at developers’ conferences, and recordings of conference talks about release management and
related topics available on the Internet), and interviews (semi-structured conversations with participants with a key role in release management, in the ecosystem, or both). The combination of studies, data sources, and analysis enabled us to triangulate and validate the results.

To understand the release process in software ecosystems better, we conducted the case study on two FLOSS software ecosystems: GNOME and OpenStack. The GNOME project is a platform to build applications for the desktop for Linux and Unix-like systems. The OpenStack project is a set of software tools for building and managing cloud computing platforms for public and private clouds.

We selected GNOME because it is a large software ecosystem [104], it is old (16 years since its announcement), it has been studied before [43, 71, 91, 143, 165], its official release is a single product composed 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. It is successful because the project has been able to follow a well defined release schedule. Because of this, the project gained credibility [108] and helped other projects that adopted GNOME to have well defined release schedules as well. For example, Ubuntu, one of the most popular Linux distributions, was originally built using GNOME on top of Debian. They were able to deliver a version every six months, because its main component (GNOME) had a predictable schedule.

We selected OpenStack because it is a young project whose first release was in 2010. It has attracted the interest of industry with over 200 companies involved in the project [135]. Like GNOME, its official release is a single product comprised of many independent projects, and it has a successful and stable release schedule of six months. Both projects have different governance models, GNOME has limited the direct influence of companies, whereas OpenStack embraces the influence of companies. Thus, both projects have similarities and differences that can enrich a study.

Although success does not imply quality [28], success has been used as an indicator of quality in the related literature because of the lack of a good metric for success [108].

To gain an understanding of their communication and coordination activities, in each ecosystem, we performed an exploratory study to identify the communication channels used by developers and the purpose of each channel, and identified the main one employed for coordination activities. We also learned the organization structure and governance model of each ecosystem to understand the context in which developers of the projects studied work better.

We followed a grounded theory [24] approach to analyze the discussions in the main
communication channel. In grounded theory, instead of starting with a pre-conceived statement, concept or theory, the researcher extract themes through manual analysis of data, which are refined in an iterative process. Through manual analysis we uncovered abstract discussion themes. In grounded theory, to obtain the themes, researchers start labeling (or coding) openly the data. From that researchers can extract concepts or themes. We followed Creswell’s guidelines [25] to label the data. We identified the leader of a discussion, the topic, and the actual purpose of the mail. After reading a set of discussion threads, the main labels started to emerge.

Thus, the analysis process of mailing discussions was:

1. Retrieval of discussions for further inspection. We retrieved the mailing lists archive data sets using MLStats [136], a tool to gather mailing archives from web sites, parse the messages, and store them in a database. We tabulated the mail archives metadata in a spreadsheet, and we added other information we collected: number of participants, duration of the discussion, period in the release cycle where the discussion took place, and the person who initiated the discussion.

2. Labeling of discussions for each email thread. In the spreadsheet we assigned a code or label to each discussion based on the subject and insights obtained when we read a set the thread of discussions. Whenever a subject was unclear or new, we reviewed the content of discussion, and add a note for future reference. In some cases, a new label emerged.

3. Cluster the codes or labels to extract topics or themes. We revisited the codes or labels to keep them consistent, and clustered them into categories of communication and coordination. These categories were the topics or themes that represent a group of discussion threads.

To validate our findings, we performed additional analysis, determined key developers, and conducted interviews. These interviews included questions that we used to obtain additional insights and to clarify any doubt emerged in the analysis process. We used interviews because in case studies the interviews play an important role to interpret the results; Walsham [164] describes it as one of the best mechanism for a researcher to reach the interpretation that participants have with respect to actions, events, views, and aspirations.

The validation process was:

1. Determine key participants as candidate to interview. We performed a social network analysis to identify key participants in the discussions, who served as potential
developers to interview. Through the organizational structure and documentation available, we also determined the developers who perform release management tasks and project leaders. In OpenStack, documentation available was abundant regarding to project leaders and their role in the release management process; therefore, the social network analysis was secondary.

2. Recruit interviewees. We recruited the interviewees based on their importance in the discussions and their roles in the release management process. In OpenStack we recruited the participants by email. In GNOME, we recruited them in person during a conference.

3. Validate our findings through interviews with key developers. We discussed our findings with developers who participated actively in the communication channel studied. The interviews consisted in three major parts: use of communication channels, roles performed in the ecosystem, and interaction with other developers. We used both open questions to gather their thoughts to compare later, and presented the discussions themes for conceptual validation.

Finally, for each ecosystem we extracted a set of lessons learned. We used theses lessons to compare both ecosystem and how they approach the release management process, and how communication and coordination process in each of them.

1.3 Contributions

In this dissertation, we explore the communication and coordination that takes places in FOSS ecosystem in order to deliver an integrated release. The contribution of this dissertation is threefold: (1) an empirical study of release management in two FOSS ecosystems (2) a set of lesson learned from the case studies, and (3) a theory of release management in FOSS ecosystems. We summarize each of these contributions as follows.

**One empirical study of release management in two FOSS ecosystems.** This empirical study deepens our understanding of the release management practices in two FOSS ecosystem. Empirical studies aim to investigate complex real life issues where analytic research might not be enough [141]. In particular, empirical software engineering aims to understand the software engineering discipline by treating software engineering as an empirical science [131]. FOSS ecosystems are complex organizational structures
that pose challenges. As a FOSS ecosystem evolves, the Release Team must: (1) be able to negotiate and reach consensus among projects and teams of volunteers, for who the Release Team have no power (2) ensure that it can handle a growing number of projects and inter-dependencies (3) monitor unplanned and uninformed changes, and (4) evolve and adapt the release process to ensure a cohesive release.

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) a successful Release Team requires both, good technical and social skills (2) an ecosystem needs a common place for coordination (3) a Release Team needs members with a variety of backgrounds (4) a Release Team needs to follow the main communication channels used by developers (5) a well defined schedule helps the Release Team in the coordination process (6) a delegation of release management tasks helps reach the whole ecosystem, and (7) a Release Team must be willing to redefine the official release as needed.

A Theory of release management in FOSS ecosystems. The theory that encapsulates our understanding of the communication and coordination regarding release management in FOSS ecosystems can be summarized as: (1) the size and complexity of the integrated product is constrained by the release managers capacity (2) the release managers should be capable of reaching the whole ecosystem, and (3) the release managers need social and technical skills.

1.4 Structure of the Dissertation

Following this introduction, this dissertation proceeds as follows. Chapter 2 summarizes the related work that is relevant for this dissertation. It provides an overview of software ecosystems, release management, and the social aspects involved in software development, such as coordination and communication, channels used for these tasks, the governance in FOSS; and, how social network analysis has been applied to study FOSS projects and ecosystems. Chapter 2 also describes and provides context of the FOSS ecosystems selected as case studies for this dissertation. Chapter 3 and Chapter 4 present the case studies that were used as the foundation for our theory, in each case study we discuss the methods used for our empirical work. Chapter 5 presents the main contribution of this work, which is our theory of communication and coordination for release management in FOSS ecosystems. The theory is built upon the literature described in Chapter 2, and on the case
studies discussed in Chapter 3 and Chapter 4. Finally, Chapter 6 summarizes the major findings of this dissertation and a description of the problems which would benefit from further research.
Chapter 2

Background

“As a software developer, I envy writers, musicians, and filmmakers. Unlike software, when they create something, it is really done — forever. A recorded album can be just the same 20 years later, but software has to change.

Software exists as part of an ecosystem, and the ecosystem is moving.”

—Moxie Marlinspike

This chapter contains a review of the related literature and background support of this dissertation. This chapter covers the topics related to the research questions and the case study subjects and the relevant studies on software ecosystems, release management, social aspects in software development, and GNOME and OpenStack. It also covers secondary yet relevant studies on governance in FOSS, communication and coordination in FOSS projects, software ecosystems in business, release management in industrial settings, and release management from an operational research perspective.

2.1 Software Ecosystems

There are multiple aspects of a software ecosystem than can be studied, depending on the research focus. As a consequence, there are several definitions and classifications of software ecosystem. This section presents an overview of the definitions and their context. It also presents a set of classifications that some researchers have applied to focus their research. Finally, the rationale of the software ecosystem concept used in this dissertation.

1“Reflections: The ecosystem is moving”, https://whispersystems.org/blog/the-ecosystem-is-moving/.
2.1.1 Definition of Software Ecosystem

The concept of software ecosystem has been defined by several authors, whose definitions vary depending on the research context in which the studies have been conducted. In the literature, we found the same concept—although with different meanings—applied in areas related to both business and software engineering.

Within the business context, a software ecosystem has been defined as a derivation of “business ecosystem” introduced in 1993 [118]. Kittlaus and Clough define software ecosystem as “an informal network of (legally independent) units that have a positive influence on the economic success of a software product and benefit from it” [73].

Another definition, in the business domain, is provided by Messerschmitt and Szyperski (as cited in [98]): “a collection of software products that have some given degree of symbiotic relationships.”

Jansen et al. define software ecosystem as “a set of businesses functioning as a unit and interacting with a shared market for software and services, together with the relationships among them. These relationships are frequently under-pinned by a common technological platform or market and operate through the exchange of information, resources and artifacts” [70]. This definition is widely used in research of software ecosystems in the context of business planning [98], where we can distinguish two major aspects: first, research specific to business that involves interaction between stakeholders, independent software vendors, support chain, alliances with business partners; and second, studies in large private corporations, for example, through the analysis of requirements elicitation, something that rarely happens in FOSS ecosystems.

Bosch defines software ecosystem as “[a] set of software solutions that enable, support and automate the activities and transactions by the actors in the associated social or business ecosystem and the organizations that provide these solutions” [17].

Finally, Lungu et al. define software ecosystem as “a collection of software projects which are developed and evolve together in the same environment” [90].

In summary, there are multiple definitions of software ecosystem, each one with a specific research scope. Therefore, the definition to use in a given research will depend on its research scope.

2.1.2 Research Scope of Software Ecosystems

Among the definitions of software ecosystem, Goeminnie and Mens [104] distinguished between ecosystems in-the-large and ecosystems in-the-small, regardless of its size. The
former is similar to Jansen’s definition of software ecosystem, which refers to a set of actors that interact in a shared marked [70]. The latter, oriented to its internal structure, defines a software ecosystem as a set of software projects that evolve together [89, 91, 90], share infrastructure, and are themselves part of a larger software project. Our research is focused in the inner workings of a large software project.

Knauss et al. [74] argued for 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 [17] (2) business and managerial perspectives [70, 69], and (3) FOSS ecosystems [89, 144]. The focus of this dissertation is FOSS ecosystems.

In FOSS, software ecosystems are composed of multiple individual projects, although they might be invisible for a user of such software. For example, a typical GUI desktop system is composed of a file manager, text editor, email client, web browser, window manager, general settings manager and the underlying libraries to build applications. All of them are expected to work as a single integrated system, even if each one is developed independently. Each might have its own release cycle, yet it needs to coordinate with the other parts of the large scale software ecosystem to properly function as a whole.

Previous research on software ecosystems has focused on improving the software development process. The work of Goeminne and Mens in this area started by exploring a set of potential research questions for further investigation that could lead to an improvement of the software development process and to assess the quality of FOSS projects [47]. 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 other developers that are needed to articulate the tasks in software development [104]. Through a framework for quantitative analysis of software ecosystems, Goeminne and Mens analyzed the organizational structure of GNOME, and determined the subdivision of community, their activities, and how these communities evolved over time [48].

Vasilescu et al. [162] studied the workload across projects, and across contributors in the GNOME ecosystem, and introduced the concept of **ecosystem community** to emphasize that studying contributors was as important as studying the contributions. Thus, **ecosystem community** is defined as the “collection of all contributors to the projects in the software ecosystem”. The study of contributors involves the study of the companies for which they work for, which overlaps with the concept of studying ecosystems in-the-large.

German et al. [45] explored the email traffic in the R ecosystem, and studied the corre-
2.2 Release Management

The aim of this dissertation is to further the understanding of communication and coordination in software ecosystems in the context of 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 [108] 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 dissertation addresses 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 [37] 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 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.

2.2.1 Release Management Strategies

Rossi [140] claimed that there are three strategies to manage the release process:

- **Feature driven development.** The criteria to release is based on the completion of a set of features that developers consider important.

- **Time based development.** The criteria to release is based on a scheduled date set well in advance.

- **Quality based development.** The criteria to release is based on a minimal quality that the features implemented must have before delivering the product.
Figure 2.1 depicts these three strategies to manage the release process, where only two of them can be fulfilled at once, and release managers must prioritize which ones they will focus on [140].

![Figure 2.1: Triangle of strategies to manage the release process. Only two of them can be fulfilled at once, unless there were unlimited resources.](image)

To release on time with a given quality, a project must sacrifice features. If the project aims to release a set of high quality features, then it must accept delays. Finally, if a project prioritizes delivering a set of features on time, it might be at the expense of the product’s quality.

Through a series of interviews to FOSS developers (core developers and release managers), Michlmayr [108, 113] identified the first two strategies (feature and time), leaving implicit the third one: quality. 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.

As Michlmayr [109] reported, several projects (for example, Debian, GNOME, gcc, Linux kernel) have migrated from a feature-based release process to a time-based one to make their releases predictable. Feature-based releases are associated by some developers as “release when it’s ready”, and with long delays, because there might be features to add or bugs to fix. In contrast, Time-based releases enforces the deadline to ship a piece of software by omitting the features that are not ready [109, 110]. 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 [109].
2.2.2 Release Management as an Optimization Problem

Release management has been also studied as an operation research problem. Some of these optimization problems have been approached from the perspective of software reliability and risk assessment (to release when the software is reliable enough). Others have studied this problem with a focus on how to assist managers regarding where and when to allocate resources [76, 85].

In the literature, there are two classes of development models:

1. Conditions analysis to determine when to stop the development and testing [139]. Models assume an initial number of independent failures that can be triggered with a given probability; the goal is to maximize the time to operate the software while minimizing the probability of failure.

2. Cost-benefit analysis to determine when the benefits of releasing early will surpass the costs of fixing bugs [125, 169, 76]. Models evaluate when the cost of testing and repair an error before release exceeds the cost of damage and repair after release [76], and the benefit or damage to an organization derived once the software is released. The benefit depends on the number of customers of each version.

Regardless of the class, models assume certainty when errors would be fixed, and therefore, their corresponding cost can be estimated. The costs considered are fixed costs (documentation, distribution, training), costs of fixing an error, cost of improving the software, and the cost of opportunity due to obsolescence and life-time of the software. Some of them, also assume that fixing an error does not introduce regressions.

For the purpose of our research, it is difficult to study release management as an optimization problem for several reasons. The main one is that software development is also a social activity, which consists of interactions and understanding between individuals. The tasks can be partitioned but require coordination that cannot be interchanged [18]. Unlike manufacturing, the software development is a creative process with unique characteristics that make productivity different from one developer to another [78], and therefore, it is hard to standardize and control. Additionally, the main interest of this study is the coordination between the projects within a FOSS ecosystem where the management of a project is directed by the same developers. In general, in FOSS ecosystems there are no project managers that decide where to put the effort, and therefore, it seems hard to “allocate resources”. Similarly, it would require a redefinition of what benefit means, as in FOSS
ecosystems it is unclear that the software release would bring direct economic benefits to the ecosystem.

In summary, the questions we are trying to answer are beyond the scope of an optimization problem, because FOSS ecosystems does not have a clear method to allocate resources and costs, and benefits are difficult to quantify. Although possible, it is the coordination process the one we are interested in this study, which consists in building consensus to reach agreements, in order to release a cohesive product.

2.3 Social Aspects in Software Development

Software development is more than writing code, it involves a set of technical and social aspects that researchers recognize that software engineering must take in consideration [105, 32]. In distributed settings, the software development process requires additional work to overcome different strategic and cultural views to design, implement, and test software [58, 55]. Empirical studies in industrial settings report that software development takes longer in distributed teams, as “cross-site communication and coordination issues are critical to achieving speed in multi-site development” [58]. Communication and coordination is challenging in geographically distributed teams, which is the nature of many FOSS projects [113]. In a FOSS ecosystem can be even more challenging to handle the communication across multiple teams and projects to integrate their projects and make a coordinated release.

Previous research on communication and coordination in software ecosystems has focused in a temporal analysis of information flows [75], and then obtained a structural map about flows between actors [74]. However, to our knowledge the requirements and challenges that release managers face in software ecosystems have not been explored.

2.3.1 Communication and Communication Channels

On their media richness theory, Daft and Lengel [31] argue that organizations process information to reduce uncertainty and ambiguity. Uncertainty is the absence of information. An organization with high uncertainty requires to answer more questions and to learn more information to reduce the uncertainty. Ambiguity occurs when the same piece of information may have multiple conflicting interpretations. High ambiguity in an organization means confusion and lack of understanding.

The capacity to process a given amount of information helps reduce the uncertainty. In
media richness theory, uncertainty can be reduced by providing sufficient information, and ambiguity can be reduced by providing rich information [31].

When taking decisions, people work under constraints like time to process data and amount that are able to rationalize. An item of information is richer than another if it can reduce ambiguity quicker.

Communication channels vary in their capacity to provide richer or leaner information. Examples of rich communication channels are face-to-face and video interactions. They are rich because these channels enable immediate feedback, the information can be checked, and provides 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 [82]. Leaner communication channels are effective to process standard data and well understood messages, however, they may require rules and procedures; for example, netiquette.

There is a second source of uncertainty that is produced by the need of integration between multiple teams or projects within an ecosystem. As Daft and Lengel state: “people come to a problem with different experience, cognitive elements, goals, values, and priorities” [31]. When the difference between teams and projects is small, but the interdependency is high, then the coordination can rely on leaner communication channels because the ambiguity is low. When the difference is high, then rich communication channels can help reduce ambiguity. The frequency of communication will depend on the interdependence between them. The higher the dependency, the higher the coordination needs.

In FOSS development, Michlmayr and Fitzgerald [109] reported that the parallel and independent nature of FOSS development reduce the amount of active coordination needed. However, regular synchronization between the different teams and projects is useful for awareness of changes and for reducing potential conflicts.

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 [134]. 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 [134] call “richness media paradox” because the rich media can simultaneously improve and impair the communication.
Thus, the use of rich communication channels should be considered when the attention, motivation and immediate feedback of participants is key. Lean communication channels enable participants deep thought and deliberation to process information, giving the receiver time to think, elaborate and discern [82].

FOSS development teams use multiple communication channels. For those FOSS projects developed by groups of people distributed across the globe, richer communication channels may not be available when required. Therefore, there is a prevalence of certain channels over others, depending on the projects and the resources available to them.

Among the communication channels used in FOSS projects, mailing lists and IRC are the most frequently used according to Fogel, German et al., and Gutwin et al. [41, 45, 53]. Mailing lists are used as public forums for asynchronous communication whereas IRC is used as instant messaging for synchronous communication.

2.3.2 Communication on Mailing Lists

Mailing lists have attracted the attention of researchers in the last decade, possibly because mailing lists archives are publicly available for a wide range of FOSS projects. The relevant studies on mailing lists are described in the next paragraphs.

Because mailing lists have been declared as one the main communication channels in FOSS development, researchers have focused in determining communication patterns in mailing lists. Guzzi et al. [54] focused on the Lucene project; the study determined that only 35% of development discussion threads correspond to implementation of code artifacts, project developers participate in 75% of the discussions, and that other communication channels may play an important role. In a similar study, Izquierdo et al. [67] studied the mailing lists of the 50 most active projects in Sourceforge; the study reports a high correlation between the number of contributors and the traffic in the mailing, however, they could not find any interesting communication pattern. In a study of the Python project, Barcellini et al. [6] manually analyzed the discussions held on a mailing list\(^2\) and studied the role of each participant in the discussions with respect to their influence in the decisions; the study reported the types of activities that take place in such mailing list. Unlike these studies that investigated mailing lists, in this dissertation we narrowed the scope of our study: first we looked for mailing lists used for coordination and communication within an ecosystem, and then we looked what kind of discussions were held on them. Therefore, the communication patterns and outcome of these studies differ.

\(^2\)The python-dev mailing list.
German et al. [45] analyzed mailing list archives to understand the evolution of user and developer community of the R ecosystem. In R, there are core and user-contributed packages. However, building a community around those packages differ depending on the origin of the package: core or user-contributed. The timing varies from months for core packages to a year for user-contributed packages. The study consisted in using regular expressions to analyze the mailing list traffic corresponding to each type of package. In contrast, we looked a common place that gathers developers from different projects of an ecosystem that coordinate towards producing a major product that integrates those multiple projects.

Ibrahim et al. [66] studied the most important factors that influence a developer to contribute to a thread. They applied Naive Bayesian and Decision Tree classifiers to the mailing list archives of three projects: Apache, Postgresql, and Python. Based on these classifiers, they built personalized models to identify in which threads a developer would participate based on previous contributions. The study reports that the most important factors are: (1) length of the thread (long thread discussion increases the odds of a developer to participate in a discussion) (2) developer activity in the list (recent participation increases the odds of getting involved in more discussions), and (3) the message content (subject and body).

Bohn et al. [16] performed content-based social network analysis on two mailing lists of the R ecosystem. Content-based social network analysis consists of combining text mining with social network analysis. First, a “communication network” is created with relationships between participants (who answers to whom). Second, a “interest network” is created with relationships between participants and the topics they participate in. For example, two participants are connected if both have an active participation in a discussion containing a given term (topic). The study found that the shared interests can only be determined for highly active participants, as the more an individual participates, the more data is available to extract preferences in their participation. Additionally, to find relationships between participants interested in a topic, the study suggests to use only the email subject as the content is prone to contain noise.

Bacchelli et al. [4, 3] linked email contents to software artifacts, and subsequently they classified the content into five categories: text, junk, code, patch, and stack trace. These studies arrived to similar conclusion than Bohn et al. [16]: using the subject field might be enough considering the noise found in email messages.

Bernardi et al. [11] studied the topics discussed in reported issues in two similar projects: Firefox and Chrome. By analyzing the communication happening in the reporting of is-
sues, they tried to determine differences and similarities in the communication between both projects. To identify topics, they applied Latent Dirichlet Allocation (LDA) [15] to the content of each reported issue, an iterated over the results to remove duplicates. They used the coherence metric introduced by [86] to measure the quality of bug reports. They found that the discussions were heterogeneous and there was a non-negligible overlap in some topics. Based on the level of noise found in emails reported by Bacchelli et al. [4, 16, 3], the analysis of emails should be focused on their subject field, rather than the whole message.

2.3.3 FOSS Governance

According to Markus [99], FOSS governance is “the means of achieving the direction, control, and coordination of wholly or partially autonomous individuals and organizations on behalf of an [FOSS] development project to which they jointly contribute”. 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 [99]. FOSS projects can encompass a variety of methods or process for developing software, which can vary according to the type of governance model they have [12]. Berkus [10] identified five types of FOSS projects related to governance style:

Solo. The majority of the software development is performed by one or two developers.

Monarchist. Projects that start as a solo project, but they evolve and develop a large community, which are ruled by a benevolent dictator. Linux and Perl are examples of monarchist projects.

Community. The software development is performed by a significant number of developers, who run the project democratically, and the decision making is reached via consensus. They might have a steering committee to resolve disputes and set direction. In any case, the steering committee is formed by members of the community. Postgresql and Debian are examples of this type of project.

Corporate. The software development is own and lead by a private company. Projects of this kind are likely to have a “dual” licensing model. MySQL and BerkeleyDB are examples of corporate projects.

Foundation. A Foundation is a formal organization that manages the project. The Foundation can be the liaison between developers and companies interested in the project.
Apache, GNOME, and OpenStack are examples of projects managed by a Foundation.

2.3.3.1 FOSS Foundations

In the remaining of this section, we further explain the Foundations because the ecosystems we present on this dissertation are governed by a Foundation. In particular, we focus on Foundations registered in United States because: (a) The Foundations that govern the ecosystems we studied are registered there, and (b) Hunter and Walli [64] reported that many FOSS Foundations are registered in United States, which make such Foundations relevant in the context of this study.

In United States, FOSS Foundations are non-profit organizations. However, there are two major type of FOSS Foundations defined by their goals and legal tax status: (1) charitable organizations whose goal is public good, and (2) business league or trade associations whose goal is the members benefits.

The charitable organizations are regulated by the Section 501(c)(6) of the Internal Revenue Code\(^3\). These organizations represent the project and can receive donations (that are tax deductible), which are used to cover the expenses of the organization, and sometimes to fund totally or partially developers or projects. This type of Foundation is usually chosen by FOSS communities because the public good is complementary to their philosophy [64]. Examples of this type of Foundation are: the Apache Foundation, the Software Freedom Conservancy, and the GNOME Foundation.

The business leagues or trade associations are regulated by Section (501(c)(6) of the Internal Revenue Code\(^4\). This type of organization is chosen by a collective of vendors interested in collaborate on a project while keeping balance control of it [64]. Examples of this type of Foundation are: the Eclipse Foundation, the Linux Foundation, and the OpenStack Foundation.

Foundations can help create a safe environment for FOSS projects to exist. Aside the legal framework, a Foundation can offer multiple services to a FOSS project, such as:

- **Intellectual property management.** Foundations can take care of the management of brand, copyright, patents, or any type of intellectual property used for the benefit of the project.

---

\(^3\)https://www.irs.gov/charities-non-profits/charitable-organizations

• **Fund management.** Foundations can manage a bank account to collect donations and fees. Similarly, they can decide how to spend those funds wisely and equally.

• **Technical infrastructure management.** Foundations can make sure a project have the infrastructure to operate, for example, servers to host source code repositories, issue trackers, mailing lists, or any type of service.

• **Representative governance.** A Foundation can provide an organization that guarantee that the active participants take the decisions.

In FOSS, Foundations exist to support projects by providing a legal structure, governance, and intellectual property management. They are usually established when a FOSS project is growing or have the potential of growing.

### 2.3.4 Social Network Analysis in FOSS Projects and Ecosystems

In this section, we explain how social network analysis can be used to study and understand FOSS projects, and how other researchers have applied it to relate source code repositories, developers, mailing list and other data sources. We are interested in networks based on developer activities because they could provide us with insights of key actors in the communication that we might overlook otherwise.

Wagstrom *et al.* [163] studied the interaction between developers, and how a community of developers around a FOSS project would evolve over time. To that end, they performed simulations of FOSS ecosystems using social network analysis from multiple data sources: blogs, Advogato\(^5\) (a social networking site) and mailing list archives of three projects. The analysis consisted of multiple data sources to obtain a more accurate representation of developer interactions. For mailing lists, they considered that two developers were linked if one of them responded to the other. For blogs, they considered that a developer was aware of another if they linked a blog post of the latter. Advogato is a social networking web site where developers certify the level of expertise of other developers, establishing a network of trust among them. Advogato provides information about projects that developers enter manually. That information was used to perform the social network analysis. The study determined that mailing list data was the best data source to estimate the size of a community and the interaction between participants, and therefore, to study the social structure of a FOSS community.

\(^5\)http://advogato.org
To understand the relationship between the communication and coordination activities of developers in the Apache HTTPD project, Bird et al. [13] applied social network analysis combining both email and source code activity. They found a strong correlation between email activity and source activity: developers with a high participation in the source code through commits also have high participation in the mailing list studied. This work is based on a single project and a single developer mailing list. We are interested in study the communication and coordination between projects. In a subsequent study, Bird [12] applied different social network metrics to other projects, such as, Apache, Perl, Python, Postgresql, and Ant.

López-Fernández et al. [92, 93] applied social network analysis to source code repositories and characterized projects by interpreting different social network measurements. They defined two type of networks: developers and modules (projects). Thus, two developers are linked if both committed code to the same project. Similarly, two modules (projects) are linked if a developer has committed code to both modules. Based on those definitions and interpretations they did a case study of Apache, KDE and GNOME. However, the study considered each project individually, not as part of an ecosystem.

Lungu et al. [90] 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. The study of the ecosystems is beyond the goal of the case studies presented in this section, as they focused on social network analysis as a technique to study source code repositories.

Ogawa [124, 123] has worked on the visualization of interactions of people in projects, and how to visualize developers with respect to their participation in one or more projects [122]. These visualizations show developers who change the same source code files within a time frame as interactions.

Martínez-Romo et al. [100] studied the collaboration between a FLOSS community and a company (Ximian) supporting two FOSS projects: Evolution and Mono. The goal of the analysis was to identify the efficiency of the network structure based on the average coordination degree. They found opposite results for each project. In the development of Evolution, the researchers found that Ximian reached a higher community involvement, which was shown in a strong network structure. However, the network structure in Mono was deficient and the authors concluded that Ximian did not reach the same level of involvement of external contributors to the project.
2.4 The GNOME Project

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 as a desktop operating system. 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 [43]. All of these components highly integrated that result in a common product: the GNOME Desktop.

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—The GNOME Desktop—a non-profit organization was created in 2000: the GNOME Foundation [43]. 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 [153]. 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 [154].

The GNOME Foundation’s Board of Directors receives input from an Advisory Board. The Advisory Board comprises 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.

The GNOME project has been widely studied, especially within the Mining Software Repositories (MSR) research community. In 2009 and 2010, GNOME was the case study of two mining challenges in the MSR conference. In 2009, the challenges were: (1) to demonstrate the usefulness of mining tools to find insights within the GNOME ecosystem⁶, and (2) predict the code growth of each project. In 2010, among the two challenges, one

---

⁶Although it was not required to study multiple projects in the ecosystem.
was partially related to GNOME: to demonstrate the usefulness of mining tools in finding relationships between version control systems, software packages for distributions, and discussions in mailing lists between GNOME, FreeBSD and Debian/Ubuntu [62].

Lungu et al. [91] focused on the visualization of the source code activity in the GNOME ecosystem over time. Through the visualization, the study distinguished three phases in GNOME’s lifetime: (1) introduction (from 1998 to 2000 there were few projects with low activity) (2) growth (from 2000 to 2003 the activity of two major projects overshadowed the others), and (3) maturity (from 2003 to 2010 there are no outliers in the activity and the activity peaks are related to the GNOME release cycle). They also found three patterns in developer’s involvement: (1) there is no developer active from the whole period studied (2) some contributors are active in a short period of time and then disappear, and (3) some contributors arrive to the project in “groups”, they start and leave the project about the same time.

Neu et al. [120] created a web tool—Complicity7—to visualize software ecosystems at different abstractions levels: from individual projects and contributors to the whole ecosystem. Each level of visualization is based on basic metrics, such as number of commits, number of source lines code, number of projects, and number of contributors.

There are studies on the workload of contributors and projects in the GNOME ecosystem. Vasilescu et al. [162] determined that the workload varies depending of the type of contributor. For example, translators tend to commit less frequently but to broad number of projects; whereas programmers tend to commit frequently to a small number of projects. Koch and Schneider [77], and German [43] reported that the distribution of workload with respect to file modifications is left-skewed, where few developers contribute most of code. Casebolt et al. [22] compared authoring with respect to the file size. The study suggests that large files are likely to be authored by one dominant contributor. In contrast, the author dominance is likely to be more spread in small files.

Walters et al. [165] report a tool—OSTree—for continuous integration in the GNOME ecosystem. The tool builds the latest versions of each module of the GNOME Desktop, bundle into a testable system ready to be downloaded and run. The tool helps the Release Team and other developers in GNOME to test the latest snapshot of the GNOME Desktop regularly.

There have been studies also on GNOME’s communication channels. For the studies on mailing lists, see Section 2.3.2. Studies on other channels include Internet Relay Chat (IRC), GNOME’s issue tracker (Bugzilla), and blog posts. which are detailed below.

---

7http://complicity.inf.usi.ch
Shihab et al. [148] mined the meeting logs of the GTK+ project (a core library in the GNOME ecosystem) held on IRC. The study compared the attendance per meeting over time, and the discussions’ involvement of the participants in each meeting. The result is that the number of participants in the meetings has grown, and there is a high level of participation in the discussions. In the study there is no analysis of the content, only attendance and participation.

Several studies have focused on GNOME’s bug database (Bugzilla), whose purpose have been: (1) predict the bug severity (2) determine quality of bug reports, and (3) determine the efficiency of developers to address issues. The prediction of bug severity has been done using two approaches: Lamkanfi et al. [81] used the Naïve Bayes classifier to predict the severity of a bug with an accuracy between 70% and 80%; and Linstead and Baldi [86] explored the use of Latent Dirichlet Allocation (LDA) to analyze the bug content, but without conclusive results. Two studies have focused in the influence of the quality of a bug report and likelihood of getting the issue addressed [5, 145]. Luijten et al. [88] compared the efficiency of individual projects to address issues within the GNOME ecosystem, finding that some projects are more efficient than others.

Pagano and Maalej [128] used LDA to explore how developers communicate through blogs. GNOME was studied, along with Eclipse, Postgresql, and Python. 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.

2.4.1 Cross-cutting Teams

The cross-cutting teams are teams who contribute to multiple projects within the GNOME ecosystem. 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. Like project teams, each cross-cutting team has its own internal structure and decision making process, and is autonomous with respect to other teams.

2.4.2 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.

2.5 The OpenStack Project

OpenStack was started in 2009 by Rackspace, as part of a project for NASA to produce software to run cloud computing services. Once the project grew, RackSpace created a non-profit Foundation in 2012\(^8\) to promote OpenStack, to oversee its development, and as a mechanism to encourage other companies to contribute to its development.

From an organizational point of view, OpenStack is a federation of projects, each developed independently. Like in GNOME, only a subset of projects hosted in the OpenStack infrastructure are considered part of the “official” release, the common product is named OpenStack.

The OpenStack Foundation responsibilities include overseeing its software, promoting the project, building a community and legal affairs (such as brand protection). Specifically\(^9\): (1) enable developers to contribute code easily while taking care of legal matters. (2) manage events to bring the community together globally. (3) manage legal affairs, like brand protection and Contributor License Agreement (CLA) process. (4) enable educational resources to help evaluate, deploy, and develop of OpenStack. (5) promote the OpenStack brand (6) promote the ecosystem of companies that build businesses with or around OpenStack. (7) perform analysis and report the “State of OpenStack” in multiple aspects, such as jobs outlook, and economic impact of OpenStack.

In the OpenStack ecosystem, the projects are categorized into six types depending on their maturity and purpose (see Table 2.1 for a summary of each type). A core project is one that is part of the “official” release, provides a service to the user and is mature. An incubated project is one that provides a service and has not reached the maturity level to become a core project. Some projects provide a common functionality useful to more than one project, these projects are libraries within the ecosystem. The projects that are part of the infrastructure to test and build the official release, such as release management tools,

---

\(^{8}\)https://openstack.org

\(^{9}\)https://wiki.openstack.org/Governance/Foundation
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>Projects that are part of the official release of OpenStack. They have full access to branding and assets</td>
</tr>
<tr>
<td>Incubated</td>
<td>Projects on an official track to become core projects. They have partial access to the OpenStack brand and assets</td>
</tr>
<tr>
<td>Library</td>
<td>Library projects are directly or indirectly used by core projects</td>
</tr>
<tr>
<td>Gating</td>
<td>Projects that are part of the continuous integration mechanism for core projects</td>
</tr>
<tr>
<td>Supporting</td>
<td>Additional projects necessary in producing the other official projects. For example, documentation and development infrastructure</td>
</tr>
<tr>
<td>Related</td>
<td>Unofficial projects with no rights to use the OpenStack brand and assets or project resources</td>
</tr>
</tbody>
</table>

Table 2.1: Types of projects in the OpenStack ecosystem.

are categorized as gating. The documentation for end users as well as for new contributors are categorized as support. Finally, other projects that are neither core or incubated, but aggregate services that could be useful to the OpenStack ecosystem are designated as related; these projects are unofficially associated with OpenStack.

Because “OpenStack” is considered a “strong” brand within the OpenStack Foundation, only two types of projects can use it: core and incubated. In exchange, these projects must ensure “openness, transparency, commonality, integration, respect of release deadlines and facilitation of downstream distribution”.

The OpenStack Foundation is composed of individual and corporate members. Corporate members are categorized into different levels depending on their contributions. The major levels of involvements have limited spots available. Corporate participation in the OpenStack Foundation is segregated. Thus, there are eight platinum members and up to 24 gold members at any given time.

To become an individual member there is no pre-requisite of previous contribution. In the 2014 annual report, OpenStack reported 18,387 individual members [157]. However, the OpenStack Foundation distinguishes between regular members and active technical contributors. An active technical contributor is an OpenStack Foundation member who has committed a change to any of the official OpenStack projects during the latest release cycle.

OpenStack also has attracted the attention of researchers. There are two curated data sets available for researchers: one from the Gerrit Code Review system [56], and another composed of source code repositories, issue tracking, mailing lists discussions, and code review [52].
González-Barahona et al. [50] studied the participation of companies in OpenStack and compared them against another project (WebKit). The study reported three research areas to investigate: (1) activity (how companies participate in source code, discussions in mailing lists, and tracking issues) (2) neutrality (how the ecosystem is affected by companies’ own goals), and (3) collaboration (how companies work together to contribute features and solve issues in common areas). The study also presented a visualization tool to explore each research area.

In a further study, González-Barahona et al. [51] used OpenStack as a case study to retrieve and analyze Gerrit Code Review information. The study proposed a mapping of abstractions from code review to issue tracking. Thus, the same tools and knowledge on issue tracking could be applied to code review.

Robles et al. [135] studied OpenStack to create a simple estimation model to measure the effort in software development activities. The purpose of the model was twofold: first, create a simple yet sound estimation model; and second, use the model to offer a reasonable prediction. OpenStack was chosen because it has a large code base and a high number of developers working “full-time” on it (over a thousand of authors affiliated to companies).

Texeira et al. [152] explored the collaboration between companies in the OpenStack ecosystem through a study that combined mining software repositories, social network analysis, and qualitative research. The study used OpenStack as a case study to explore the role of groups and sub-communities within OpenStack.

Thongtanunam et al. [161] studied the code review activities in Qt and OpenStack to determine the chain of responsibility in the source code (known as code ownership), which comprises the developers who authored a piece of code as well as the developers who reviewed the code. The study used Qt and OpenStack because of these project have large repositories (modules) where every change in the source code had undergone a code review.

2.6 Summary

In this research, we refer to a software ecosystem as defined by Lungu et al.: “a collection of software projects which are developed and evolve together in the same environment” [90]. Furthermore, we narrowed the scope of study to the internal structure of the ecosystem as a set of software projects that evolve together [89, 91, 90], share infrastructure, and are themselves part of a larger software project. In words of Goeminne and Mens [104], a study of ecosystems in-the-small.

To study coordination for release management in FOSS ecosystems is necessary to
consider social and technical factors. There is a variety of communication channels with different media richness information, a variety of contributors distributed geographically with different interests, and multiple independent projects to coordinate in order to release an integrated product. We provided an overview of the related literature regarding to these concepts, which mainly have studied separately or in other contexts.

We also introduced the FOSS ecosystems we chose to study: GNOME and OpenStack. In the following chapters, we present two case studies of release management in FOSS ecosystems, we compare them, and then propose a theory that explains our understanding of release management in FOSS ecosystems.
Chapter 3
Case Study: The GNOME Ecosystem

“We need to communicate to make sure at the end 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

In this chapter, we report the case study of the GNOME ecosystem we conducted to understand the communication and coordination in the release management process of a FOSS ecosystems better. We describe the communication channels used and their purpose within the ecosystem, the type of communication performed by developers, the key developers in the release management activities, and the lessons learned from the study. Specifically, we aim to answer the following research questions:

Q1. In the context of release management, how do developers in the GNOME ecosystem communicate and coordinate to build and release a common product based on different projects?

Q1.1. What are the communication channels used for release management?

Q1.2. How do developers communicate and coordinate for release management?

Q1.3. Who are the key actors in the release management process?

Q2. What are the release management tasks in the GNOME ecosystem?

Q3. What are the challenges that release managers face in the GNOME ecosystem?

We first describe the study design in Section 3.1, the process to select the communication channels, the criteria to collect and clean the data, the analysis performed, and the use of interviews to triangulate and enrich the findings. Then, in Section 3.2, we explain
the release process in the GNOME ecosystem. In Section 3.3, we report the findings per each research question. We found that the Release Team (a) sets the release schedule that guides the coordination activities in GNOME (b) coordinates with projects and cross-cutting teams, (c) faces challenges associated with the size and complexity of managing multiple independent projects, and (d) relies on building consensus based on technical merit to overcome the lack of power over developers. In Section 3.4, we discuss the findings and present the lessons learned we extracted from studying GNOME. Finally, we present the threats to validity of this study and the conclusions.

3.1 Study Design

To answer the research questions, we used a mixed methods approach [35] that employs data collection and analysis with both quantitative and qualitative methods [25, 141, 171, 172]. The goal of the study was twofold: first, to deepen our understanding of the release management coordination in the GNOME ecosystem, and second, to triangulate and enrich our findings through interviews with developers.

The study was composed of five steps, we (1) identified the main communication channel used to coordinate projects and teams within the ecosystem (2) collected and cleaned the communication data from that channel; we also gathered data from source code repositories to obtain a better understanding of the discussions and their participants (3) analyzed the data collected and extracted discussion themes (4) conducted a social network analysis on the mailing list to determine the key participants in the discussions, and (5) conducted interviews to triangulate our findings and obtain additional insights from developers.

3.1.1 Communication Channel Selection

To identify the communication channels had for release management, we gathered information from the GNOME’s website. The two main communication channels recommended by GNOME were mailing lists and IRC. We focused on mailing lists, as they are archived and publicly available. We did not find evidence that communications over IRC were archived by GNOME, which makes the historical analysis harder, if not impossible.
3.1.2 Data Collection and Cleaning

At the time of this study, the GNOME ecosystem used 285 mailing lists archived\(^1\). We searched for mailing lists used for cross-project communication and release management. The selection criteria was based on mailing list names and descriptions, number of participants, frequency of use, and the lists recommended by the Release Team to its new members. We found that the Release Team recommends to its team members to follow two mailing lists (desktop-devel-list and release-team); such recommendation is to help new Release Team members to grasp background information about the development process within the ecosystem [155]. We contrasted the lists recommended by the Release Team against the data we collected from all the mailing lists.

We identified the Desktop Development mailing list\(^2\) as the main channel for communication related to release management: it is where the discussions of the desktop and platform development take place. To study the communication across several releases, we retrieved 32 months of data spanning from January 2009 to August 2011. We used ML-Stats [136] to retrieve the mailing list archive and split them into threads. We found this period interesting because it comprises five release cycles, including the transition between two major releases—from the series 2.x to 3.x. 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 two participants per discussion.

To associate multiple email addresses with a single individual, we used an approach similar to that used by Bird \textit{et al.} [13]. We created clusters of similar identities and then manually processed them. 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, we merged email addresses ending in \texttt{gnome.org} that had the same user name (for example, we merged in \texttt{jhs@gnome.org} email addresses like \texttt{jhs@cvs.gnome.org}, \texttt{jhs@src.gnome.org}, and \texttt{jhs@gnome.org}).

3.1.3 Analysis

We followed a grounded theory [24] 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 discus-

---

\(^1\)https://mail.gnome.org

\(^2\)https://mail.gnome.org/mailman/listinfo/desktop-devel-list
sion threads. We followed Creswell’s guidelines [25] for coding and abstracting recurring patterns.

Figure 3.1 depicts a message from a discussion and the observations noted by a researcher. We identified the leader of a discussion, the subject, and the actual purpose of the mail. After reading a set of discussion threads, the main labels started to emerge.

Ever since automake 1.9, automake has been spewing garbage like this when you try to build any module that uses gnome-doc-utils:

http://bugzilla.gnome.org/show_bug.cgi?id=507336

Discussions on IRC indicate that we pretty much require GNU make all over the place in Gnome, including for intltool. I have had numerous bugs filed and fixed against gnome-doc-utils for non-GNU systems (mostly by the venerable Joe Marcus Clarke). I can only assume that, at this point, I'm not causing any *real* problems for non-GNU systems; otherwise, I'd be seeing real bugs.

It seems to me that, if we just require GNU make, we ought to just pass -Wno-portability to automake. There's a bug and patch from Philip Van Hoof to do this:

http://bugzilla.gnome.org/show_bug.cgi?id=529120

Christian Persch says he's in favor, but thinks it ought to be discussion on desktop-devel-list.

So folks, let's discuss.

--
Shaun

Figure 3.1: Excerpt of an e-mail that shows the annotations performed by the researcher (in red). First, we looked the subject, then the content of the discussion whenever was necessary. We added a label based on the main topic of discussion.

To code the messages, we read the email subjects and associated a code to each thread. The code then represented the message’s topic or theme. Whenever the subject was unclear, we read the discussion thread in detail to get more contextual information, and searched in
other data sources\textsuperscript{3} for additional information that could provide us with more clues about the topic discussed. Thus, on unclear subjects, we also considered the role in the ecosystem of the person initiating a discussion, the roles of the other participants, the number of messages, the number of participants, and the time in the release cycle were the discussion occurred—from early planning to finally release a stable version. We used those details as follows:

The role of who initiates a discussion let us identify the individual’s status of a contributor within the ecosystem, and the potential motivations to bring a topic for discussion. We assumed that the intention of a message may vary depending of the sender. For example, an external user might bring an unfamiliar subject to the mailing list that may or not belong to the regular topics in the mailing list. On the opposite side, a regular developer, project maintainer or team member bringing an unfamiliar subject may be interesting and we would search more information about it. We determined a role by reading wiki pages of projects and teams, and by checking contributions in source code repositories.

The number of messages in a discussion was used to prioritize their analysis. We studied first and in more detail the discussions with more messages.

The number of participants in a discussion was also used to prioritize the analysis of discussions. Like number of messages, the ones with most participants were studied first and in more detail.

The time in the release cycle when a discussion occurred let us contextualize the discussion under study. This allowed us to notice repetitions of certain topics according to the stage in the release cycle when the discussion occurred.

The role of other participants in a discussion let us obtain clues about the type of discussion a participant would become involved with (that is, in conjunction with the discussion content). By separating the role of other participants and the initiator, we could distinguish among people who reply to regular developers or to newcomers in the mailing list. Also, whether developers would participate in subjects of their own expertise or in broader discussions, and thus, that could help us to label the subjects.

\textsuperscript{3}Wiki, websites, related bug reports when a link or issue identifier were available in the discussion, and source code when a commit would be referenced.
We clustered codes into topics or themes of communication and coordination. Later, we validated these themes through interviews with the corresponding developers.

Figure 3.2 depicts a summary of the email subjects, their corresponding theme given by a researcher, the number of messages and duration of each discussion. The actual spreadsheet that aggregated the discussions, label and themes has more data than shown in the figure, for example, the URL to go to the discussion thread as needed, the leaders of each discussion.

![Figure 3.2: The same idea than in the first step, but summarized and aggregated in one single spreadsheet. It is shown the subject, the theme, number of messages, and duration of the discussions.](image)

We explored the use of LDA to classify the email subject field as one of the potential lines of study, in a similar way than reported in other studies (see Sections 2.3.2 and 2.4). However, in our tests the corpus of the each document (email subject field) was too small, the generated clusters were unclear and not useful for our analysis.

### 3.1.4 Social Network Analysis

We conducted a social network analysis of the mailing list data to identify, first, the key participants in the discussions, and second, their relationship with the GNOME’s Release Team. With this analysis we unveiled the participation of the Release Team in the discussions, and identified potential candidates to interview later—we wanted to interview key developers rather than random participants in the mailing list discussions.

We built the social network using the following method: nodes represent participants of discussions. Two participants are connected if one of them replies to the other [13, 12]. An edge between two nodes represents—undirected—communication between two participants. The size of the node is relative to the number of messages sent or received.
Our goal in this step was to identify key participants in the discussions and their relationship with the release management process. Therefore, we started with an approach that let us visualize the influential nodes in the social network. We explored the social network using Gephi [8]. We applied the ForceAtlas algorithm [68] provided by Gephi, which is similar to the Früchtermann Rheingold algorithm [121]. This algorithm spatializes small-world networks with an emphasis on layouts to support analysts interpreting the graph. It pushes influential nodes to the center and less influential nodes to the border. The visualization was useful to locate the release team members in the social network, and obtain an overall view of their influence.

To verify the most influencing nodes in the visual exploration, we calculated both the degree centrality and eigenvector centrality of each node in the network. The degree centrality was used to verify who were key participants in discussions based on the numbers of messages sent or received by a developer, and the eigenvector centrality was used to verify the influence of certain developers in the social network. Although both the degree centrality and eigenvector centrality measure the influence of a node in the network, we preferred the eigenvector centrality because it favors the nodes connected to other nodes that are themselves central within the network.

In addition, we looked for gatekeepers in the social network. A gatekeeper is “an individual who is located in a communication structure so as to control messages flowing through a communication channel” (as cited in [42]). A gatekeeper enables or connects two individuals in a communication channel. We were interested in exploring if the Release Team members, or any other developer, acted as gatekeepers. To determine gatekeepers between developers we calculated the betweenness centrality of each developer (node). The higher the betweenness, the higher the potential of an actor to be a gatekeeper [146].

3.1.5 Interviews

The purpose of interviewing developers was twofold: first, to triangulate our findings, and second, to enrich our findings with additional insights of the development practices and release management process. We conducted ten semi-structured interviews with GNOME developers who had participated in the discussions we had studied.

3.1.5.1 Recruitment

We created an initial list of candidates for interview based on whether they would meet at least one of the following two criteria: (1) importance in the social network, and (2) being
a member of the Release Team. As we explained in Section 3.1.4, we determined the importance of participants in the social network by a visual exploration, and by calculating the degree, betweenness, and eigenvector centrality of each developer.

We considered that the most important nodes in the network were developers who were highly involved within the GNOME ecosystem, and therefore, more likely to attend to GNOME’s main conference, the **GUADEC**. GUADEC is where GNOME developers get together to discuss the state of the project and its future. During the conference, we were able to interview ten—out of the top 35.

### 3.1.5.2 Structure of the Interviews

Interviews consisted of three parts: (1) inquiry about roles in the project and communication channels our interviewees used (2) to comment on our findings, and (3) to comment on specific interactions based on the social network chart we presented. The Appendix B.1 contains the script we used to conduct our in person interviews. Below we provide details of each part.

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. To enable us to engage in the interviews better, we fa-

---

4http://www.guadec.org/
5Their importance in the social network.
miliarized ourselves with the type of discussions that each of these developers participated, and selected the ones we considered may bring us richer answers (in case we needed to remind the interviewee their interactions).

3.2 Background: The Release Process in GNOME

In this section we report the release management process in GNOME, which provides an immediate context to our findings. As we will see later, many aspects of the communication flow in GNOME is tied to the release cycle followed in GNOME.

3.2.1 Release Management and the Release Team

The importance of release management in GNOME is shown in the mission of its Foundation described in its charter, where it is addressed the delegation of GNOME’s operational management [154]. As described in Section 2.4, the Board of Directors of the Foundation is responsible of releasing GNOME. Originally, the Release Team was appointed by the Board of Directors, trying to represent the entire GNOME community. Over time, the Release Team has evolved independently. When a position on the Release Team is vacant, the team itself decides who should fill it. It is currently composed of nine individuals.

The objectives of the Release Team are: (1) to define the requirements of GNOME releases (2) to coordinate and communicate with projects and teams, and (3) to ship releases within a predefined quality at a specific time. 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 (it has never happened). The GNOME Foundation members are encouraged to participate in the discussions to affect the decisions of the Release Team. In extreme cases of disagreement, members can propose a referendum [154].

3.2.2 The Release Cycle

 GNOME releases follow a time-based release cycle of six months (26 weeks). A release cycle starts immediately after a major release. It starts with a release meeting to evaluate the last cycle, and to discuss the next cycle [155]. At this point, the Release Team creates a schedule of activities to deliver a release on time. This schedule is used by the Release Team to keep the teams informed of the milestones, verifying that 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].”

Figure 3.3 illustrates the release cycle. The development of a GNOME release begins immediately after the release of the previous stable version. This stable version is kept in a branch for bug fixes. Regular development continues in the main branch.

During the first 20 weeks of a release cycle, there are development releases every four weeks [156]. In the 21th week, the 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 update the manuals. When the second freeze starts, a new restriction is added on top of the previous ones: it is forbidden to change any string 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 is focused on testing and fixing critical bugs, no changes can be made without the approval by the Release Team. In week 26, the final version is released and a new cycle begins.

### 3.3 Findings

We structure this section based upon the research questions. As we divided research question $Q_1$ into three questions, we answer each of the sub-questions separately. Where appropriate, we provide quotations from interviews to illustrate our findings.
3.3.1 How do developers in the GNOME ecosystem communicate and coordinate to build and release a common product based on different projects?

To find out how developers communicate and coordinate in the GNOME ecosystem to create a single common product, we: (1) identified which communication channels they use (2) determined the topics of their discussions, and (3) identified the key actors in the communication.

3.3.1.1 What are the communication channels used for release management?

In the GNOME ecosystem, developers communicate and coordinate using several communication channels: email, IRC, face-to-face, wiki, blog, and bug tracker. 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.”

The Release Team may monitor a variety of communication channels to have multiple sources of information that could be relevant to a release. However, the Release Team prioritizes in four of them. Specifically, the guide for new Release Team members [155] recommends participating in three mailing lists (release-team, desktop-devel-list, and devel-announce-list) and one IRC channel (#release-team).
In practice, we found that the Release Team keeps track of multiple channels as well, with different grades of importance. Table 3.1 shows a summary of each communication channel. Each communication channel is detailed in the following paragraphs.

### Main communication channels

<table>
<thead>
<tr>
<th>Channel</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mailing lists</td>
<td>They are used for discussions about projects. Two mailing lists stand out as for coordination of the whole ecosystem and for internal discussion of the Release Team: desktop-devel-list and release-team, respectively.</td>
</tr>
<tr>
<td>IRC</td>
<td>An interactive Internet text messaging system. In GNOME this is used as instant messaging for synchronous communication.</td>
</tr>
</tbody>
</table>

### Other communication channels

<table>
<thead>
<tr>
<th>Channel</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bugzilla</td>
<td>A web-based bug tracking system, used to track and discuss bug reports or feature requests that are reported by users and developers.</td>
</tr>
<tr>
<td>Wiki</td>
<td>A collaborative tool used in GNOME as the organization space for development and community engagement. It keeps the information of the release process, and instructions for developers to make releases.</td>
</tr>
<tr>
<td>Blogs</td>
<td>A web-based aggregator of GNOME developers’ blogs.</td>
</tr>
<tr>
<td>Conferences</td>
<td>General meetings where developer meet in person to discuss the development of GNOME. There is one annual conference since 2000.</td>
</tr>
<tr>
<td>Hackfests</td>
<td>Focused face-to-face meetings of developers to accelerate the work on a specific project.</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of communication channels found in the GNOME ecosystem.

- **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 GNOME ecosystem. One of the main uses of the mailing list is to discuss topics that require more thought, as a developer stated:

  “Sometimes I use mailing lists for discussing major things... [B]igger changes like [changing] several related things [are] easier to layout in an overview email, where you really describe what you want to achieve, how you are going to go about it and what things are going to be affected.”

The Release Team gives special importance to the global list desktop-devel-list, even though its description—“GNOME Desktop Development List”—seems unrelated to release management. Desktop-devel-list is the mailing list where developers from different projects converge to discuss about GNOME in general, and it is the one
used by the Release Team to communicate and coordinate with the whole GNOME ecosystem. As another developer stated:

“... from the stand point of project health, ... it is good that important decisions are documented in mailing lists.”

The Release Team also uses an internal mailing list (release-team) to discuss and decide issues directly related to release management. Decisions taken in the internal mailing list are later announced and discussed in desktop-devel-list for general awareness and obtain feedback. Some topics discussed broadly by the GNOME developers in desktop-devel-list can also be discussed here, in order to settle a common view of the Release Team. As one Release Team member stated:

“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.”

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

• IRC. IRC is an interactive chat system. Similar to mailing lists, in GNOME there are internal and global chat channels. The Release Team holds meetings once or twice per release cycle using an internal channel (#release-team). The internal channel is also used for discussions within the team and for developers to get quick answers about release management.

The Release Team also monitors the #gnome-hackers channel, which is the IRC equivalent of desktop-devel-list mailing list, where developers from different projects converge and communicate within the ecosystem. Due to the limitations of IRC, such as the requirement to be connected and online to keep a conversation, it is less inclusive than the mailing list. As one developer stated:

“IRC is good to have a conversation between two or three people. But if you [need] high level coordination that affect the entire project that tends to be on the mailing lists.”
However, it is useful for quick coordination between developers working on the same set of features, solve questions, or any topic that might be of the general interest of the GNOME ecosystem. As another developer stated:

“If people are already involved in working on something, IRC works very nicely for coordination.”

Some discussions related to release management might arise in #gnome-hackers, which may be addressed in the same channel or moved to #release-team to get the other Release Team members attention. Therefore, the Release Team monitors #gnome-hackers to be aware of what developers discuss that might affect a release, or that might need some clarification from the Release Team.

- **Bugzilla.** Bugzilla is a Web-based bug tracking system. In GNOME, it is used to track and discuss bug reports or feature requests that are reported by users and developers. According to one of the interviewees, Bugzilla makes it easier to trace progress, compared to other methods:

  “[Using Bugzilla] it is easier to keep track of the progress over a longer time because an IRC conversation is very transient.”

Furthermore, the Release Team and developers use Bugzilla to keep track of features and critical bugs for future releases, as a Release Team member stated:

“I keep Bugzilla in mind, see what issues are important, what should be solved for the next major release; and I take care [of] nagging developers about the development schedule for the next release cycle.”

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.

- **Wiki.** A Wiki is used as “**GNOME’s development and community organization space**”\(^6\). Every developer with an account can edit any wiki page, except the ones that have been explicitly restricted. Here the Release Team maintains information of the release process, provides instructions for developers to make releases, and provides details about the current release schedule, such as important dates. In every

\(^6\)https://wiki.gnome.org/
release cycle, the Release Team creates a draft of the schedule, requests feedback from developers in desktop-devel-list mailing list, and once it is decided, announces the final schedule. It is used during the whole development cycle. At the beginning, to edit and communicate the schedule, and later, to keep track of the major features and critical bugs, with links to discussions and bug reports. Keeping the information centralized in the wiki page, enables any Release Team member to take over a task from another member if needed. For developers, it is the reference to keep track of the schedule, as well as to inform themselves about the release process and guidelines. As a Release Team member stated:

“In each cycle, we put a draft [of the schedule] on the wiki, ask on the development list for feedback, after a while, we make a final [version] and we announce [it] ... we expect the core modules in GNOME and its maintainers to stick to it.”

• Blogs. GNOME aggregates the blogs of its developers in a common location called Planet GNOME\(^7\), which is defined as “a window into the world, work and lives of GNOME hackers and contributors”. This is done through an application that retrieves blog posts from external sites (developers’ blogs) using the Rich Site Summary (RSS\(^8\)) protocol, sorts them by time, and presents them together in one single GNOME website: Planet GNOME\(^9\).

Planet GNOME works as a news website, the audience expected is wider than the subscribers of a mailing list or the regular participants of an IRC channel, and whose entries are automatically updated from blogs posts written by developers.

In Planet GNOME there is no restriction or filter of topics that developers can post about. Although these are personal blogs, some Release Team members use them to communicate release-related decisions and to inform others about the overall status of a release. Some developers announce major changes in the software they maintain, explain new features available in libraries, share the best development practices, and express their points of view regarding the software they maintain, plans and the future of the ecosystem.

\(^7\)https://wiki.gnome.org/PlanetGnome
\(^8\)http://www.whatisrss.com/
\(^9\)http://planet.gnome.org/
As a consequence, Planet GNOME also serves as a source of information to distributors, FOSS news sites, or any other party following the GNOME ecosystem without being subscribed to any mailing list in particular.

Developers are aware that the audience of Planet GNOME is wider than the inner circle of GNOME, and the impact of their blog posts might be different from mailing lists or IRC. Thus, developers keep a healthy tone to present their views through blog posts. As one developer stated:

“On blog posts it is easier to keep the tone of the conversation healthy, because people-wise the blogs are very attached to their public image, it is more [visible] than a comment inside the blog of another person. So, when the conversation happens between blogs, within the context of Planet GNOME, it is more productive, or more detailed.”

- **Conferences.** GNOME holds an annual conference for all GNOME developers: the **GUADEC**. In this conference, the Release Team presents the state of the project. In a panel, the team discusses the future of GNOME with developers. The conference is an opportunity for the Release Team to have face-to-face meetings, and to obtain direct and immediate feedback from other developers within the ecosystem. The GNOME Foundation may fund totally or partially the travel expenses of Foundation members who request it. Members who are speakers or are required to participate in meetings get higher priority for funding than only attendants. Exceptionally, and depending on the importance for the project, non members can receive travel assistance.

The importance of conferences is beyond the merits of communicating on technical matters, it is also social because it allows developers to understand how each developer communicates better. As a Release Team member stated:

“Face to face meetings, which doesn’t happen very often, [they are] really useful because you have two things: (a) socialize, not just talk about your issues and what you need for the projects, which is essential; [and] (b) put a face on a voice or a face on a character. The bandwidth is much higher.”

Face-to-face meetings is a rich communication channel (higher “bandwidth”) that allows developers to know each other better, which is also helpful for further inter-
actions on mailing lists or IRC. The benefits extend beyond the actual purpose of the face-to-face meetings, as the same Release Team member continues:

“When I met someone in real life, it always change my interaction with this person because I understand better the person... In IRC or email you don’t exactly know how the person is speaking. Once you know how the person speaks in the real life, you hear the voice when you read the email. So, you get the character. It’s not just text.”

This better understanding might help reach consensus, or at least, to comprehend each others’ positions, and possibly keep the tone of their interactions healthy.

- **Hackfests.** Hackfests are focused face-to-face meetings of developers to work on a specific project, feature, or release. Hackfests are organized either to discuss and plan or to get work done. Depending on the topic, some Release Team members are invited to participate to bring their perspective into the planning. These meetings are organized by developers, however the Foundation provides travel assistance to developers who request it based on the same criteria used for conferences. The value of hackfests is similar to conferences, as a developer stated:

  “GUADEC, and other conferences and hackfests, are very valuable for having face to face contact with people... you have a lot more bandwidth.”

**In GNOME, the Release Team uses mailing lists and IRC as the main communication channels for coordination, mailing lists for long term discussions, and IRC for quicker decisions that involve less than four people. Regardless, the Release Team might use multiple channels as input to gauge their decisions, including face-to-face meetings.**

### 3.3.1.2 How do developers communicate and coordinate for release management?

From our analysis of the `desktop-devel-list` mailing list we found it is primarily used for coordination. Nine themes of discussions emerged. Five of them are directly related to release management activities. Table 3.2 shows a summary of each category.

The detail of the themes related to release management activities are:
### Discussions related to release management

<table>
<thead>
<tr>
<th>Theme</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request for comments</td>
<td>Long-term planning and roadmap discussions</td>
</tr>
<tr>
<td>Proposals and discussions</td>
<td>Short-term planning</td>
</tr>
<tr>
<td>Announcement</td>
<td>Announce releases and create awareness of the status of each project in the ecosystem</td>
</tr>
<tr>
<td>Schedule reminders</td>
<td>Specific type of announcements to keep developers on schedule</td>
</tr>
<tr>
<td>Request for approval</td>
<td>A decision is requested from a small group of people</td>
</tr>
</tbody>
</table>

### Other type of discussions

<table>
<thead>
<tr>
<th>Theme</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events coordination</td>
<td>Organization of conferences, sprints, or related activities</td>
</tr>
<tr>
<td>Expertise seeking</td>
<td>Developers seeking people who work in specific artifacts</td>
</tr>
<tr>
<td>Knowledge seeking</td>
<td>Developer seeking specific knowledge (either technical or organizational)</td>
</tr>
<tr>
<td>Out of scope</td>
<td>Anything else</td>
</tr>
</tbody>
</table>

Table 3.2: Summary of themes found, grouped in related and unrelated to release management activities.

- **Request for comments**: These messages are 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 or major changes whose execution could take one or more releases.

These discussions start at the beginning of each release cycle, then are revisited during the release cycle. Developers state their opinions and the Release Team gauges the overall sentiment. We verified this observation in an interview, as a Release Team member stated:

> “Part of the purpose of doing these discussions [is] to figure out what people concerns are, and make sure they can be addressed.”

Examples of request for comments’ discussions are “empathy integration with the desktop” (discuss integration of instant messaging directly in the desktop and independent of an external application), “Consolidating Core Desktop libraries” (integrate classes from multiple ‘microlibraries’ into a common library), “RFC: gtk-doc and gobject introspection” (discuss how to integrate the API documentation for language bindings with the new bindings that are generated automatically).
Figure 3.4: Request for comments during the release cycle.

Figure 3.4 shows that requests for comments happen through the whole development cycle in spite that these discussions are encouraged at the beginning of the cycle.

- **Proposals and discussions:** This is a specific type of request for comments tied to a particular project, but with potential indirect impact on other projects or teams. Its nature and recurrence make it worth a theme by itself. 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 might raise concerns regarding the long-term sustainability of the external library—such as development activity, availability, or the library’s record regarding security fixes.
These are short term proposals with their focus on the current release cycle. When opinions diverge, the Release Team takes the final decision.

Examples of proposals and discussions are “systemd as external dependency” (request an external dependency to satisfy features requested by some projects), “Module Proposal: GNOME Shell” (propose a new module as part of the official release), “New proposed GnomeGoal: Add code coverage support” (propose adding metrics of code coverage to every active module in GNOME).

![Boxplot showing discussions during the release cycle](image)

Figure 3.5: Proposals and discussions during the release cycle.

Figure 3.5 shows that proposals and discussions happen through the whole cycle.

- **Announcement**: These are 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 development branch, new external dependencies, and reporting on the status of project goals.

Examples of announcements are “GNOME 3.0 Release Candidate (2.91.92) Released!” (announce a new development version), “GNOME 3.0 Blocker Report for week 09” (announce the list of pending bugs to fix before the next release).

Figure 3.6: Announcements during the release cycle.

Figure 3.6 shows when the announcements take place during the development cycle.

- **Schedule reminders:** This is a specific type of announcement that the Release Team uses to send periodic reminders about the current release cycle’s stage. The Release
Team reminds developers to release a new version, start the period of feature proposals, and so on. Its nature and recurrence make it worth a theme by itself, even though they are a type of announcement.

Examples of schedule reminders are “Release Notes time!” (reminder for developers to write about new features in their modules), “GNOME 2.29.90 beta tarballs due” (reminder for developers to release a beta version before the deadline), “Last call for comments on module proposals” (reminder for developers to provide feedback on proposals before the Release Team decides).

![6-month release cycle (milestones)](image)

Figure 3.7: Schedule reminders during the release cycle.

- **Request for approval**: These messages request the breaking of a freeze period towards the end of the release cycle, when the Release Team has taken control of the
changes, as we explain in section 3.3.2. There are three types of changes that require to break the freeze: (1) changes to translatable strings used by the applications (2) changes to the user interface, and (3) changes to the source code. The decision to accept or reject a break in the freeze period is taken by the Release Team, the Documentation Team, or the Translation Team; although the discussion is open to everyone. These requests require a timely decision as they occur close to the release date. All decisions require the approval or rejection of at least two members of the Release Team. Decisions on user interface changes also require the approval of the Documentation Team; and decisions on translatable string changes also require the approval of the Documentation and Translation Teams.

Examples of request for approval are “Hard code freeze break request for gvfs” (fixing a regular bug with a trivial patch), “[Freeze break request] gksourcerview crash” (fixing a critical bug with a trivial patch), “String change in gnome-session” (fixing a wrong description for a configuration key setting).

Figure 3.8 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.

The remaining four themes are less relevant to release management activities:

- **Events coordination**: This is a special type of announcement and further coordination related to the organization of conferences, sprints, or hackfests. An example of events coordination’s discussion is “Planning for the Boston Summit” (an invitation to developers to draft the working session at a mini-conference).

- **Expertise seeking**: These are questions from developers who seek others who work on or are in charge of specific projects, libraries, or other artifacts within GNOME. Example of expertise seeking: “Patch review for metacity: disable-keybindings” (looking for an experienced metacity developer to review a patch in Bugzilla).

- **Knowledge seeking**: These are questions that developers ask about specific technical or organizational issues. An example of knowledge seeking is “Who exactly are the freezes for primarily: upstream or downstream?” (ask clarification of the schedule freeze purpose, whether is intended for GNOME developers or distributors).

- **Out of scope**: In this theme, we collected any other messages, such as spam and generic complains. An example of an out of scope discussion is “A few notes about
**Figure 3.8: Request for approvals during the release cycle.**

"the direction in which gnome desktop is going" (user opinion about GNOME Desktop and the GNOME ecosystem).

Table 3.3 shows both, the number of discussions and messages during the period studied. We present both to balance their importance. Individually, there are less Request for comments and Proposal and discussions than Announcements. However, the proportion of messages of each of them reflects that those are the core themes discussed in the mailing list.

Overall, we noticed that discussions started by well-known developers attract responses from other well-known developers, more than discussions started by other people. To verify this observation we asked to our interviewees their criterion for getting involved in a
<table>
<thead>
<tr>
<th>Theme</th>
<th>Discussions</th>
<th>Messages</th>
<th>Median of messages per discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#</td>
<td>%</td>
<td>#</td>
</tr>
<tr>
<td>Discussions related to release management</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Announcement</td>
<td>238</td>
<td>25.19</td>
<td>740</td>
</tr>
<tr>
<td>Proposals and discussions</td>
<td>219</td>
<td>23.17</td>
<td>2,074</td>
</tr>
<tr>
<td>Schedule reminders</td>
<td>45</td>
<td>4.76</td>
<td>236</td>
</tr>
<tr>
<td>Request for approval</td>
<td>22</td>
<td>2.33</td>
<td>83</td>
</tr>
<tr>
<td>Request for comments</td>
<td>181</td>
<td>19.15</td>
<td>2,505</td>
</tr>
<tr>
<td>Other type of discussions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Events coordination</td>
<td>27</td>
<td>2.86</td>
<td>44</td>
</tr>
<tr>
<td>Expertise seeking</td>
<td>25</td>
<td>2.65</td>
<td>184</td>
</tr>
<tr>
<td>Knowledge seeking</td>
<td>151</td>
<td>15.98</td>
<td>764</td>
</tr>
<tr>
<td>Out of scope</td>
<td>37</td>
<td>3.92</td>
<td>317</td>
</tr>
<tr>
<td>Total</td>
<td>945</td>
<td>100.00</td>
<td>6,947</td>
</tr>
</tbody>
</table>

Table 3.3: Summary of discussions and messages per category, grouped in related and unrelated to release management activities, and sorted alphabetically.

discussion. Our interviewees reported that they would be more inclined to participate in a discussion started by known developers, as they already know their expertise.

The release schedule of GNOME guides the type and timing of coordination activities discussed in the main communication channel. The scope of decisions varies from project-wide to localized the more the project approaches the release. The desktop-devel-list mailing list is the communication channel where most of the cross-coordination activities take place, in there, there are two themes that stood out: proposals and discussions and request for comments, which are the themes where developer discuss the short and long term roadmap of the ecosystem.

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

As described in Section 3.1, we conducted a social network analysis to determine key developers in the release management process.

3.3.1.3.1 Social Network Analysis Results. Figure 3.9 shows the results of this analysis. We present the social network of desktop-devel-list mailing list discussions. The same
figure is split into six sub-figures to highlight the interactions of different Release Team members (labeled from Release Team #1 to #6, from Figure 3.9a to 3.9f), and how they reach different participants in the social network. The box in each sub-figure 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.

The three major nodes are the ones with more participation in the social network. Among them, the bigger one is the release-team mailing list in which only Release Team members participate. The release-team mailing list has a normalized eigenvector of 1.0, which means that this node is the most influential in the social network. In other words, the release-team list is usually added explicitly 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. With respect to the other two major nodes, Figure 3.9c) is a senior Release Team member 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 (Figure 3.9), 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 themes 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. When we shared this observation with our interviewees, they were surprised at the beginning but acknowledged the themes and agreed with the results of the analysis. It was unexpected for our interviewees because it was not the “official” purpose of the mailing list.

We found that some members of the Release Team are more prominent than others in the social network. For example, Figure 3.9f shows a localized interaction between a Release Team member with other developers, and through them it can reach other developers. In contrast, Figure 3.9c shows interaction with a wide range of developers; and in between, the other Release Team members. The difference indicates that some Release Team members interact with more people than others, and therefore, are better connected.

3.3.1.3.2 Composition of the Release Team. The focus in the discussion by Release Team members is not random: the Release Team members tend to participate in a wide range of discussions. In the social network, this would mean being connected to the ma-
Figure 3.9: Scope of interaction of different Release Team members in the mailing list social network. We selected the six more representatives interactions of Release Team members. 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.

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. We verified this tendency in the interviews. According to one Release Team member, 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. As we learned from a Release Team member in an interview:

“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.”

Because the Release Team members trust each other decisions, other members may not participate in the public discussion where another Release Team member is already participating. It is an unwritten policy, and perhaps, part of the culture of the Release Team, as the interview continued:

“... so it is a way to ... I would not say efficient, but you know that to cover a lot of topics, you do not need to involve yourself in everything if somebody else you trust takes care of that. It is not by design, it just happens.”

We also observed that members of the Release Team also contribute to other areas of the project, which are not restricted to software programming. The areas are documentation, translation, accessibility, quality assurance, system administration, and, in general, anything that needs attention. This provides the Release Team with awareness and a closer connection to different areas of the project. First, awareness is useful to address any issue as soon as possible, and second, a closer connection with different teams within the ecosystem may help approach them with more confidence, and eventually, these teams may be more willing to listen and accept the requests and decisions of the Release Team. We verified this finding in the interview, as a Release Team member stated:

“Is is little bit about peer groups, ... who works with whom. If I know that one person out of the Release Team should get a reply 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 like ‘hey, you have a better connection to this person, could you provide the answer?’ ”
Thus, the more diverse the Release Team members are, the most representative the Release Team would be of the ecosystem and the wider the coverage within the ecosystem. When a member leaves, the Release Team members discuss which areas need improvement and recruit someone accordingly. We verified this observation in the interviews, as a Release Team member stated:

“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.”

To support diversity across supporting organizations, the Release Team does not allow that more than two members belong to the same organization—directly or indirectly.

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.

3.3.2 What are the release management tasks in the GNOME ecosystem?

The tasks 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 tasks follow the release cycle. Depending on the stage in the cycle are the tasks that Release Team members perform.

With respect to the communication themes, the first task maps to Request for comments and Proposals and discussions (developers discuss proposals, the Release Team decides and makes a plan for the current and future releases); the second task maps to Schedule reminders, and Request for approval (the Release Team communicates periodically the status of development cycle and coordinates the pending tasks per milestone); and the third task maps to Announcement, Schedule reminders, and Request for approval (Release Team keeps the development on track, takes over the control of the projects, and makes releases).
The Release Team defines the requirements of a GNOME Release by starting a discussion with developers during the feature proposals stage at the beginning of the release cycle (as shown in Figure 3.10; for a detailed description of the main milestones, see Section 3.2).

To facilitate coordination, the Release Team prepares the release schedule and announces it. Based on the schedule, developers propose new features early in the release cycle. The Release Team coordinates the community, helps reach consensus, and discusses and decides the adoption of proposals. Each stage in the schedule requires different coordination and communication activities. One of the simpler tasks is sending periodic schedule reminders to keep developers aware of development cycle stages and to remind them to prepare their own projects for release. Approving or rejecting a request for comments is more complex, as a lot of discussion may be required to reach consensus.

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. This task involves building every component of the release set and validating that the software runs as expected. If a build fails, the Release Team will get in touch with the developers of the failing component and get the build fixed. Release Team members acknowledged that this is one of the most time-consuming tasks. By continuously building and testing a planned release, the Release Team monitors the quality of the product during the whole release cycle. They determine critical bugs and follow-up with developers to fix them. They also coordinate with distributors of GNOME regarding potential issues.

As part of shipping a release, 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. The release notes is a document that highlights the major changes
between one release and another. It is useful for users, distributors, integrators, and other developers to be aware of changes, and eventually, to decide the paths to upgrade to the next version.

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.

3.3.3 What are the challenges that release managers face in the GNOME ecosystem?

As described in Section 3.1, we consolidated information found on GNOME’s websites, which lead us to design our interviews. From both, we identified the five major challenges that release managers face in the GNOME ecosystem, they need to (1) coordinate projects and teams of volunteers without direct power over them (2) keep the build process manageable (3) monitor for unplanned changes (4) monitor for changes during the stabilization phase, and (5) test the GNOME release.

3.3.3.1 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 in GNOME. Projects are “owned” by the contributors who actively work on them, and these people make decisions regarding their projects. We verified this observation in the interviews, where the vision was shared between regular developers and Release Team members. According to a developer:

“Maintainers have the last word most of the time. If the conflict is about a maintainer not agreeing with your vision of what needs to be done, with specific technical decision, then it is [the maintainer’s call].”

The Release Team does not have any official power over developers. To coordinate multiple projects, the Release Team relies on building consensus based on technical merit. One challenge the Release Team faces is to convince developers of its judgment and knowledge in the release process, especially when it is difficult to reach consensus. For example,
in 2010 the Release Team considered that GNOME 3.0 was not ready yet and decided to release GNOME 2.32 instead (a maintenance release of the GNOME 2.x series); this decision was strongly opposed by a group of vocal developers and maintainers, who had the power to release any version of their own projects; however, they respected the Release Team decision regardless of their disagreement.

To earn trust from the community, the Release Team builds awareness of the whole release process, and makes the community an active participant in the discussion and decision process. A Release Team member verified this observation:

“It is difficult to coordinate well, because 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 time-based schedule facilitates this task by providing the same information to the whole community beforehand [113], and providing developers a sense of ownership of specific tasks. Therefore, the developers become more involved in the release process. This emphasizes the importance of social skills and power of persuasion of the Release Team members.

3.3.3.2 Keep the Build Process Manageable

GNOME is composed of multiple pieces of software, each one with its own set of dependencies. Dependencies can be internal and external of the ecosystem. Each of these dependencies might require their own set of dependencies. When the dependencies grow, building the whole GNOME becomes cumbersome as it takes longer and there are more points of build failures. A direct consequence of this may be less volunteers building and testing the whole GNOME before the release, and it also 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.”
One way to make the building stack smaller is to avoid managing external dependencies whenever is possible. Thus, the Release Team defines two types of dependencies: system dependencies and regular dependencies. The system dependencies are external dependencies that are mature enough that are 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.

System dependencies evolve as any software, and newer versions may provide features that GNOME developers might want to use in their projects. However, a newer version of a system dependency may require moving it to a regular dependency because of its novelty. To avoid growing the building stack, the Release Team encourages developers to delay new version requirements for system dependencies as much as possible. Once a new version requirement is imminent, it is necessary to coordinate with all affected projects early in the release cycle in order to address any possible incompatibility.

Additionally, the Release Team monitors the regular external dependencies for their inclusion in distributions. When regular dependencies become system dependencies, they no longer need to be managed and built by GNOME.

3.3.3.3 Monitor for Unplanned Changes

Changes in the Application Programming Interfaces (API) and Application Binary Interfaces (ABI) of internal and external libraries pose a challenge to release managers. The libraries that GNOME provides try to guarantee stability in both API and ABI. This means that any application developed using a public API of a stable series will continue working with future releases and it will not require recompilation (ABI stable). Because there are several libraries at different levels in the GNOME stack and these are maintained by different people, it is challenging to track unintentional breakages before a release. Once a change of API/ABI is detected, it requires coordination with the developers in charge of each part involved.

Some API/ABI changes might work well in some build configurations, but break in others. For example, some changes 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, like GTK+, we have to be worried about Windows or FreeBSD.”
Similarly, 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\textsuperscript{10}.

### 3.3.3.4 Monitor for Changes During the Stabilization Phase

During the stabilization phase, developers request approval for changes in their projects. This is an honor system that is respected by developers and not technically enforced. However, sometimes changes happen without asking for approval first, usually by mistake. These changes are noticed by contributors of the affected teams or other developers monitoring commits, who raise the issue in desktop-devel-list and ask either to clarify or revert the changes.

### 3.3.3.5 Test the GNOME Release

Given the number of projects that compose GNOME, as well as dependencies on external projects, it is hard to test the latest complete development version of GNOME. These quality assurance activities are performed by a small group of developers, mainly the Release Team, who is in charge of continuous integration. GNOME does not have a continuous integration process, yet this is something that the Release Team feels is needed. In the words of a Release Team member:

“[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. When we release four or six 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 [165] 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, as well as other developers who can run the latest snapshot without building every module.

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

3.4 Discussion

In this section, we discuss our findings and present some lessons we learned from studying release management in the GNOME ecosystem. We further discuss our findings in Chapter 5, which presents a theory that explains our understanding of release management in FOSS ecosystems.

Organization. The GNOME ecosystem is governed by the GNOME Foundation, a charitable organization whose purpose is the public benefit. The GNOME Foundation is the supporting organization that oversees the ecosystem administratively and technically. It is part of the Foundation duties to oversee the release process, however, it has delegated the technical decisions to the Release Team.

Release Team. 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.

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 ecosystems, and keeps itself aware of the ecosystem.

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\textsuperscript{11}. 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 hanged a banner during GUADEC stating “Down w/Release Team” and the symbol of anarchy. In GNOME, the maintainers of each project decide what to release, and therefore, the disagreeing developers could have released a newer version of their projects (for GNOME 3.0) regardless of the Release Team directive. However, the Release Team’s decision was respected, even though it was unpopular.

**Release Schedule.** After a period of proposals and discussions, the Release Team decides the features that 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.

**Communication.** The communication activity is tied to the release schedule. Five out of nine discussion themes in a general GNOME development mailing list were 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 discussions during the whole cycle, and publish reminders at key moments in the development cycle.

3.4.1 **Lessons Learned**

We summarized our lessons learned as follows:

**A successful Release Team requires both, good technical and social skills.** 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. They need social skills to convince de-

\textsuperscript{11}https://mail.gnome.org/archives/desktop-devel-list/2010-July/msg00133.html
velopers to perform the necessary actions to deliver the software on time, especially because they lack direct power over them.

**Need a common communication channel for coordination.** Each individual software project has its own communication channel infrastructure, such as their own git repository or mailing list. However, to coordinate multiple projects is necessary to have infrastructure that is common to all these individual projects. A common channel enables the Release Team to track the progress of all projects more easily, and facilitates the communication flow from the Release Team to the projects and vice versa. This is performed by having an ecosystem-wide mailing list for high-level discussions, and an ecosystem-wide IRC channel for quick feedback.

**A Release Team needs members with a variety of backgrounds.** The 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, 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.

**Follow the main communication channels used by developers.** The Release Team needs to communicate in a variety of ways to reach the higher number of developers within the ecosystem. Thus, the Release Team in GNOME uses electronic channels that vary from asynchronous (such as email and blogs) to more direct, interactive ones (such as IRC and face-to-face meetings). Asynchronous channels are used to reach a wider audience and enable discussions, whereas synchronous channels are used to answer quick questions, solve coordination issues that involve a few developers, and hold regular meetings. The Release Team members value face-to-face communication for both technical and social reasons. From the technical point of view, the high “bandwidth” of the face-to-face communication enables the Release Team to discuss, plan, obtain direct and immediate feedback, and get work done quicker. From the social point of view, face-to-face communication helps the Release Team
to understand how each developer communicates better; this is especially helpful to understand how individuals communicate in text based channels, such as email and IRC. 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. The main benefit of having periodical gatherings with contributors is to build awareness of the Release Team, and in general, build empathy that could enable the Release Team to solve conflicts better.

**A well defined schedule helps the Release Team in the coordination process.** 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.

### 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 [41, 43, 13, 116, 117]. 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. I manually categorized the email subject fields, and I might have introduced subjective bias in the results. I followed Creswell’s guidelines [25] 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. I extracted the topics to build the categories based on my interpretation of the subject field of each email thread. To avoid misinterpreting the actual discussions, before I performed the coding, I familiarized myself with the email threads, read some messages to obtain more contextual information, and finally, triangulated the results 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 chapter, 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 analytic generalization, which is performed by comparing the characteristics of a case to a possible target [40]. The case study presented can facilitate the analytic generalization and comparison with other cases. In addition, we conducted a second case study, which we present in the next chapter.

3.5.4 Reliability

Reliability refers to the research steps from which a study can be replicated and lead to similar results. In our study we used mailing list data that is publicly available, described the procedure to identify key developers to interview, and analyzed the data following the practices described by Creswell [25], and Corbin and Strauss [24]; therefore, We expect that other researchers would be able to replicate the study. If the characteristics of each FOSS ecosystem may differ, the underlying lessons and implications should remain unchanged.

3.6 Summary

We explored the GNOME ecosystem to gain a deeper understanding of its release management. We determined the main communication channel used to coordinate the ecosystem,
extracted meaningful discussion themes, determined the relevant actors, to whom later we interviewed to triangulate and enrich our findings.

We determined that GNOME’s developers communicate and coordinate using multiple communication channels, both asynchronous (email, blogs, bug trackers) and synchronous (face-to-face interaction in conferences and hackfests, IRC). We identified mailing lists (desktop-devel-list and release-team), and IRC as the main communication channels; mailing lists for long term discussions, and IRC for quicker decisions that involve less than four people.

We further studied the desktop-devel-list mailing list, because it comprises the discussions of the GNOME ecosystem and it is archived. We found that the scope of discussions in the desktop-devel-list mailing list vary from project-wide to localized as the project approaches the release date. The desktop-devel-list mailing list is the communication channel where most of the cross-coordination activities take place, in there, there are two themes that stood out: proposals and discussions and request for comments, which are the types of discussions where developer discuss the short and long term roadmap of the ecosystem.

Through social network analysis, we identified the Release Team as 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, as all its members have direct contact with almost all participants in desktop-devel-list mailing list, and one-degree away of every participant there. Considering that the “official” purpose of desktop-devel-list mailing list is not to coordinate activities related to release management, our interviewees were surprised by this observation, 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. The variety in the Release Team composition helps it create ties with different groups within the ecosystem. Thus, the Release Team can become more effective to communicate decisions, answer questions, raise concerns, and, in general, build trust. Our analysis of interactions could be beneficial for the Release Team, either to detect communication anomalies on time or to discard irrelevant issues faster.

We also determined the tasks of the Release Team in GNOME, which 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. These tasks follow the release cycle, and depending on the stage in the cycle are the tasks that Release Team members perform. 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. In performing the tasks, The Release Team faces multiple challenges in the GNOME ecosystem. The five major challenges we identified in this study are, they need to (1) coordinate projects and teams of volunteers without direct power over them (2) keep the build process manageable (3) monitor for unplanned changes (4) monitor for changes during the stabilization phase, and (5) test the GNOME release. These challenges are related with the size and complexity of managing multiple independent projects, developed by volunteers, in a distributed setting.

The contributions of this chapter are: (1) an empirical study of release management in a FOSS ecosystem, and (2) a set of lessons extracted from this case study. These contributions provide the foundation to build a theory of release management discussed in Chapter 5. The lessons learned from this case study are summarized as follows: first, an ecosystem needs a common place for coordination, and second, a Release Team (1) requires both, good technical and social skills to be communicate with contributors with different background, (2) needs members with a variety of backgrounds to reach the variety of teams and projects in the ecosystem, (3) needs to follow the main communication channels used by developers to address any concern that may affect the release process, and (4) use a well defined schedule as a assistance in the coordination process.

The Release Team leads the coordination efforts within GNOME—it is the glue that keeps multiple projects and teams working together towards a common goal. The Release Team is crucial for the success of GNOME, even if some of its members write little or no code at all.
Chapter 4

Case Study: The OpenStack Ecosystem

“The main trick we use to align everyone and make us all part of the same community is to have a cycle. We have regular milestones that we ask contributors to target their features to. We have a feature freeze to encourage people to switch their mindset to bugfixing. We have weekly meetings to track progress, communicate where we are and motivate us to go that extra mile. The common rhythm is what makes us all play in the same team. The ‘release’ itself is just the natural conclusion of that common effort.”

—Thierry Carrez, Director of Engineering at OpenStack Foundation

In this chapter, we present our second case study: the OpenStack ecosystem. We report the communication channels used in the ecosystem, the purpose of each channel, type of communication held in the main one, and who are the key developers in the release management activities. Specifically, we aim to answer the following research questions:

\(Q_1\). In the context of release management, how do developers in the OpenStack ecosystem communicate and coordinate to build and release a common product based on different projects?

\(Q_{1.1}\). What are the communication channels used for release management?
\(Q_{1.2}\). How do developers communicate and coordinate for release management?
\(Q_{1.3}\). Who are the key actors in the release management process?

\(Q_2\). What are the release management tasks in the OpenStack ecosystem?

\(Q_3\). What are the challenges that release managers face in the OpenStack ecosystem?

\(^1\)“The need for releases”, http://ttx.re/the-need-for-releases.html.
\(^2\)Same research questions stated in Section 1.1, but with emphasis in OpenStack.
4.1 OpenStack

OpenStack is a FOSS project for cloud computing similar to Amazon Web Services (AWS). The project started as a joint effort developed by NASA and Rackspace, and its original goals were to provide an API compatible with AWS to facilitate the migration of applications from one platform to another, and to operate on standard hardware. Over time, the project was opened to a broader audience, gained attention from the industry, and more companies joined the project. What initially was a project to provide cloud computing compatible with AWS, later was extended beyond AWS.

OpenStack abstracts the underlying hardware, consolidating the resources into three main pool of resources: compute, storage and networking. These services are operated through a dashboard, which is web interface to administrate the resources. On top of these services, developers build application services for cloud computing. Every service corresponds to one or more relatively independent projects inside OpenStack, each one with its own decision-making process.

OpenStack is an ecosystem because it is comprised of many independent projects. These projects work in coordination towards an integrated product: the OpenStack cloud platform. OpenStack has a successful and stable release schedule, it delivers a new version every six months. At the time of this study, OpenStack consisted of 49 projects, totalling 388 git software repositories. Over 200 companies participate in the development of OpenStack. Many of these companies employ developers to work on OpenStack, either full or part time. The composition of companies is broad and diverse, the same as developers working on OpenStack. The companies are Linux distribution vendors, network appliance providers, hosting providers, and integrators, to name a few. The diverse contributor base makes OpenStack an interesting ecosystem to study how the developers communicate and coordinate, and how they manage a release.

In the remaining of this chapter we present the details of the OpenStack ecosystem case study. In Section 4.2, we describe the study design: the selection process of communication channels, the retrieval and filtering of data criteria, the analysis performed, and triangulation approach of our findings through interviews. Then, in Section 4.3, we explain the release process in the OpenStack ecosystem. In Section 4.4, we report the findings per each research question: we found that (a) a well defined release cycle is essential to deliver an integrated release in OpenStack (b) cross-project coordination occurs between the Release Team and developers who represent the interest of each project (c) all together, Release Team and project representatives, have a set of responsibilities and tasks that help
them overcome the complexity of managing an increasing number of independent projects, and (d) most of them are funded by different companies, each one with its own interests. In Section 4.5, we discuss the findings and present the lessons learned from this study. Finally, we present the threats to validity of this study and the conclusions.

4.2 Study Design

We conducted a case study on the OpenStack ecosystem in a similar way than we conducted the case study of GNOME, reported in Chapter 3: we used a mixed methods approach [35] that employs data collection and analysis with both quantitative and qualitative methods [25, 141, 171, 172]. Similarly, the goal of the study is twofold: first, to deepen our understanding of the release management coordination in the OpenStack ecosystem, and second, to triangulate our findings through interviews with developers.

4.2.1 Communication Channel Selection

To identify the main communication channels used for release management, we explored the OpenStack websites, the documentation that explains to users and contributors how OpenStack works, the wiki pages that contain information about projects, teams, development processes, and the organizational structure of OpenStack. In the OpenStack wiki page we found that the three main communication channels within the ecosystem are mailing lists, IRC, and face-to-face meetings at the Design Summit (the main OpenStack conference). We found that discussions in mailing lists and IRC are archived and publicly available. To contrast the results of both OpenStack and GNOME ecosystems, we focused our research mainly in the communication in mailing lists.

In ecosystems like GNOME, many projects have a separate mailing list to hold internal discussions, and a general mailing list to hold cross-projects discussions. A developer interested in the discussions of several projects likely will be subscribed to as many mailing lists as projects the developer follows.

In OpenStack, we found 47 mailing lists archived\(^3\), most of them used for (1) local groups by region, city or country (2) infrastructure, system administration and coordination among system operators (3) OpenStack Foundation activities (4) community engagement as the point of contact of local groups with the whole community, and (5) end-user support. We found only one mailing list dedicated to the OpenStack development: openstack-dev.

\(^3\)http://lists.openstack.org/cgi-bin/mailman/listinfo
mailing list\textsuperscript{4}. Therefore, we chose openstack-dev because it is the single development mailing list for all OpenStack projects, the place to discuss the future development, and to coordinate activities of interests for release management. Furthermore, the openstack-dev mailing list is advertised as where the discussion of the next release happens:

\begin{quote}
"[openstack-dev is a] high-traffic list for the developers of OpenStack to discuss development issues and roadmap. It is focused on the next release of OpenStack ..."\textsuperscript{5}
\end{quote}

The openstack-dev mailing list posed a challenge because it comprises the discussions related to all projects in OpenStack. It contains discussions that are internal to each project, cross-project communication, and discussions of OpenStack as a whole. Even though openstack-dev is the place to discuss the development of OpenStack, it is a multipurpose mailing list (unlike in GNOME, where the mailing list we studied is mostly used for cross-projects discussions, relegating the internal discussions to other mailing lists).

As a consequence, first, we proceeded to study how the mailing worked before proceeding with filtering the data; and second, we also relied on other data sources (IRC archives, Etherpad notes, wiki pages, source code repository) to enrich the insights found in the openstack-dev mailing list discussions, and also to determine key contributors to triangulate our findings later via interviews.

\subsection*{4.2.2 Data Collection and Filtering}

In OpenStack, all projects discussions take place in openstack-dev mailing list. In this section, we clarify Mailman—the program to manage mailing list, and how it works.

The OpenStack developers rely on a feature of Mailman (the mailing list server software) called topics to distinguish discussions. A person only needs a subscription to a one single mailing list, but needs to be registered to as many topics as the developer is interested to, making Mailman topics work like a double-subscription process on the same mailing list. Mailman distributes every message tagged with a topic only to the subscribers of such topic. However, it is up to the sender to tag properly an email. Unless someone advertise the topics in a discussion or in a different channel, only the subscribers know the actual topics defined in the mailing list.

\textsuperscript{4}http://lists.openstack.org/cgi-bin/mailman/listinfo/openstack-dev
\textsuperscript{5}https://wiki.openstack.org/wiki/Mailing_Lists
In Mailman, the topics are defined by the mailing list administrator. The sender of an initial discussion must tag manually the message with a topic, making the process error prone. In practice, a topic consists of a keyword in the subject, which must be enclosed between brackets, as annotated in Figure 4.1 ([openstack-dev] is the name of the list and does not provide information because it is present in every subject). The topics can be concatenated, for example, the email subject “[openstack-dev] [ceilometer] release for Folsom” correspond to a discussion of the Ceilometer project, and “[openstack-dev] [ceilometer][horizon] release for Folsom” corresponds to both Ceilometer and Horizon projects. In the example, a subscriber of any of both Ceilometer or Horizon will receive the message.

![Figure 4.1: Excerpt of an e-mail that shows an example on how we identified the topic specific to a given project.]

Mailman topics is a mechanism to target specific subscribers, but it is up to the sender to “tag” an email with a proper topic in the email subject. Because the email subject is free form text, the sender can introduce spelling mistakes or improperly tag the topic. For example, the subjects “[openstack-dev] horizon - extra-specs questions” and “[openstack-dev] Horizon style question” refer to the same project (Horizon), but the Mailman filter to detect topics will fail and, therefore, consider it as message for the ecosystem instead of the Horizon project.

Because messages incorrectly tagged may appear to be cross-project discussions, we had to determine which messages were actually cross-project discussions and filter the data accordingly. To determine cross-project discussions, first we obtained the list of topics to separate individual project discussions from the general discussions, second, from the resulting set of general discussions, we separated the messages incorrectly tagged to leave only those whose purpose likely meant to be a general discussion. The remaining set contained the cross-project communication discussions. The detail of these two steps were as follows:

1. In Mailman, the list of topics is restricted to the subscribers. We subscribed ourselves to the OpenStack Development mailing list to identify and retrieve the list of topics
defined. We found 36 topics, each one defined with a simple regular expression that can also be used by subscribers to create custom filters in their email clients. For example, the regular expression defined for the project *Heat* is as follows:

```regex
.*\[Heat\].*|.*\[heat\].*
```

Some topics were promising, but unfortunately there were not messages matching such topics at the time of the study; for example, the topics "[Release]" and "[Cross-project coordination]" were defined, but with not matching messages. We learned that those topics were defined recently (at the time we collected the data) to be used later.

To filter out the regular project discussions from the general discussions, we built regular expressions to extract the topics from each email subject. We used the following heuristic to build the regular expression:

(a) A topic can be a word or set of hyphenated words

(b) A subject can contain from zero to multiple topics, each topic enclosed between square brackets, and each topic may or may not be separated by spaces

(c) ‘Openstack-dev’ is not a topic, but the mailing list name

(d) Consider a topic anything between square brackets at the beginning of the subject or after the term ‘Fwd: ’

(e) Consider a subject without a topic a potential cross-project discussion

(f) Consider the topics ‘general’ and ‘common’ as cross-project discussions

After this step, we narrowed the set of cross-project messages from 24,643 messages (grouped into 7,640 discussions) to 15,254 messages (grouped into 2,853 discussions).

2. Our heuristic is limited to the text that follows a fixed pattern, and mimics Mailman filters. To identify and discard the remaining messages incorrectly tagged from the general discussion set, we read the content of the discussions as needed while labeling them, and omitted them for further analysis.

In the second step, we filtered out manually 768 messages grouped into 171 discussions.
We used MLStats [136] to retrieve the mailing list archive data sets. We retrieved data from May 2012 (beginning of the mailing list) to July 2014. In total, this accounted for 24,643 messages grouped into 7,650 discussions. Once we filtered the data as described above, the cross-project communication accounted for 14,486 messages grouped into 2,682 discussions.

4.2.3 Analysis

As in Chapter 3, we analyzed the discussions in the openstack-dev mailing list following a grounded theory [24] approach. To extract themes or topics, we manually analyzed the emails of each discussion thread, we segmented them, and then labeled them with a term. We refined the terms to extract the themes or topics that represent the discussion threads better. We followed Creswell’s guidelines [25] for coding and abstracting recurring patterns.

Figure 4.2 depicts a message from a discussion and the observations noted by this researcher. We identified the subject of the discussion, the leader, and purpose of the mail. For the purpose of this example, we also identified that the message is specific to a project (this message corresponds to the Nova project).

We coded 2,853 discussions by reading both the subject and content of each email. Each code represented the discussion’s topic or theme. When the subject was clear, we skimmed the content to verify that the subject was representative of the content. For unclear subjects, we read the discussion thread in detail, and when needed, searched in other data sources for additional information of the discussion as well as the participants. Additionally, we also considered the role in the ecosystem of the people involved in the discussions, the number of messages, the number of participants, and the time in the release cycle where the discussion occurred—from early planning to finally release a stable version.

Once we finished coding the discussions, we clustered the codes into twelve themes of communication and coordination whose detail we present in Section 4.4.1.2. To facilitate the qualitative analysis of the mailing list we built a custom application, which is described in Appendix C.

4.2.4 Social Network Analysis

In the GNOME case study, we conducted a social network analysis to determine key contributors to interview to triangulate our results and to obtain further insights. In OpenStack, however, the roles in the coordination of the Project Team Leaders and Release Team is
Figure 4.2: Excerpt of an e-mail that shows the annotations performed by this researcher. First, we looked the subject, then the content of the discussion whenever was necessary. We added a label based on the main topic of discussion. The excerpt also shows an example on how we identified the topic specific to a given project.

well documented. Therefore, we determined the contributors to interview from such documentation.

In OpenStack we found that the leaders of each project within the ecosystem (known as Project Team Leaders or PTLs) engage directly with the Release Team to discuss the release management process that affects the ecosystem.

We conducted a social network analysis of the mailing list to compare the results against the GNOME case study, if the results were relevant. We also used this analysis to identify the project leaders and Release Team members in the interactions.

As we explained in Section 4.2.2, the OpenStack development mailing list contains internal discussions of its projects, and general cross-project discussions. Therefore, we performed the analysis in the subset of discussions that belonged only to cross-project communication and coordination. Otherwise, we may have obtained connections between developers for discussions within a single project.

In the social network created for our analysis, a node represents a participant in a discussion. Two participants are connected if one of them replies to the other [13]. An edge between two nodes represents—undirected—communication between two developers.
4.2.5 Interviews

We interviewed developers to triangulate our results and to enrich our findings with additional insights of the release management process in OpenStack as well as their development practices. We conducted 6 open-question interviews with OpenStack developers who had participated in the discussions we had studied.

4.2.5.1 Recruitment

The list of candidates to interview had to meet one of the following criteria: (1) being a Project Team Leader, or (2) being a member of the Release Team. We invited 48 developers who met the criteria, and 6 of them accepted to be interviewed.

4.2.5.2 Structure of the Interviews

The interviews were conducted via email, stating explicitly we reached each interviewee because of its involvement as Release Team member, Project Team Leader, or both. The interviews consisted of three parts, on which we asked about: (1) communication channels usage (2) the roles taken by developers within the ecosystem with respect to the projects they contribute to, and (3) the communication, coordination, and conflict resolution across projects. The Appendix B.2 contains the script we used to conduct our interviews. We tailored our interviews based on our previous experience interviewing GNOME developers, and we also reduced their length because written interviews may take longer to complete than oral ones (GNOME interviews were in person). Below we provide details of each part.

First, we asked each interviewee questions regarding the use of every 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.

Second, we asked each interviewee about their roles in the projects they were involved, how their roles influenced the use of certain channels, and to comment on communication patterns. We provided brief examples of communication patterns we found.

Finally, we asked each interviewee questions about coordination and planning across projects, and how they take decisions when there were disagreements between developers in charge of different components in the ecosystem.
4.3 Background: The Release Process in OpenStack

In this section, we aim to provide context to our findings by explaining the release management process in OpenStack. As we will see later in this chapter, some aspects of the communication flow in the OpenStack ecosystem are related to the release cycle.

4.3.1 Release Management, Release Team and Project Team Leaders

The release management is important to OpenStack, the OpenStack Foundation has the position of Release Manager among its permanent staff. The Release Manager is in charge of the coordination of the OpenStack release management with each project and the infrastructure required to keep the release process under control.

At the beginning of this study, the release management in OpenStack was related only to the core projects, which are the ones that are part of the OpenStack release. The incubated projects had to follow the release process, and they must successfully release at least two milestones before they would be considered to become a core project\(^6\). Towards the end of the study, the release management process dropped the concept of “official release” and shifted to a tagging system, where each project is tagged according to its adoption, maturity, and age in the industry (we further discuss it in Section 5.3.5).

The Project Team Leader is the maintainer of a project and has three main roles: (1) make the final decisions related to the project (2) be the liaison between the project and the Release Team, and (3) be the point of contact for the project. The Project Team Leader can delegate any of the last two roles to an active contributor of the project. Each project has only one Project Team Leader, who is elected every six months among the active contributors of the project. In OpenStack, an active contributor is someone who has contributed in the last six months.

The Release Team and the Project Team Leaders define the release cycle, and create the schedule of activities for each cycle. They keep the projects informed of this schedule, verifying that activities are completed as required in order to deliver releases on time.

4.3.2 The Release Cycle

In the OpenStack ecosystem, each project must deliver a new release under one of three different release cycles [158], which are: (1) common cycle with development milestones (a

\(^6\)https://wiki.openstack.org/Governance/Approved/Incubation
time-based release cycle which results in a single release at the end of the development cycle) (2) common cycle with intermediary releases (a time-based release cycle which results in multiple releases within the development cycle), and (3) independent release cycle (a release cycle without a coordinated release). Additionally, there are two types of projects: managed and independent projects. Managed projects are those projects that request to the Release Team to coordinate their release process, and they follow one of the first two release cycles. Independent projects are those projects who coordinate their release process themselves, and they are free to choose any release cycle.

The OpenStack common cycle follows a time-based release cycle of approximately six months (27 weeks). A release cycle starts immediately after a major release. The first two to three weeks are for planning and session preparation for the major semi-annual OpenStack conference: the Design Summit. This release cycle includes three development milestones, which are reference points in time to organize the development cycle. Projects can set specific deadlines that match those dates. The time between milestones or releases may vary because the Release Team proactively avoids conflicts with major holidays and conference schedules. Overlaps between release cycles may exceptionally occur; for example, a version in development may be released after the next version has started its cycle.

The common cycle with intermediary releases is a subset of the common cycle with development milestones, where the development milestones are replaced by early stable releases. The libraries are the projects that traditionally follow the common cycle with intermediary releases. Once the libraries are considered stable (free of major architectural changes), there is a reduction in the number of changes in general, therefore, they may release earlier for the benefit of the projects that depend on them.

Figure 4.3 illustrates the common cycle with development milestones. The development of an OpenStack release begins immediately after the release of the previous stable version. The source code is divided in two branches: the stable version branch (for bug fixes) and the main branch where the development for the new release happens.

Towards the first milestone (M1), the development is characterized by completing the proposals\(^7\) and discussing about them. The second milestone (M2) comprehend normal development work, and to indicate the time when developers should start the stabilization of the development work leading to the third milestone (M3). Once the cycle reaches the third milestone, the development of new features stops (“feature freeze”), and client libraries must deliver final releases. Two weeks after, all projects issue a first release candidate. From then onward, projects cannot introduce changes in messages that need translation.

\(^7\)The proposal are known as lightweight specifications or blueprints.
This freeze guarantees stable user interfaces to translators and technical writers, so they can work polishing the translations and documentation, respectively. Other two release candidates follow, and developers work on stabilizing the software. After the third release candidate, only emergency last-minute release candidates are allowed. In week 27, the final version is released and a new cycle begins.

4.4 Findings

To make the comparison easier, we structured this section in the same manner than the GNOME case study: each research question is followed by its respective answer. We also provide quotations from interviews when appropriate to illustrate the findings better. The quotations correspond to a selection of opinions we gathered from developers that in our opinion provides a good context for our findings, and the triangulation performed.

4.4.1 How do developers in the OpenStack ecosystem communicate and coordinate to build and release a common product based on different projects?

To find out how developers communicate and coordinate in the OpenStack ecosystem to create a single common product, we: (1) identified which communication channels they use (2) determined the discussions themes, and (3) identified the key actors in the communication.

To understand our findings better, it is important to notice that in OpenStack there are
multiple stakeholders from the Release Team point of view: (1) developers (any collaborator involved in the OpenStack development) (2) Project Team Leaders (maintainers or developers in charge of a project) (3) Release Team members (developers who participate actively in the release process) (4) everyone else (users, integrators and any individual interested in OpenStack).

4.4.1.1 What are the communication channels used for release management?

Table 4.1 shows a summary of each communication channel we found. Each communication channel is described in detail in the following paragraphs.

- **Mailing lists.** In OpenStack there is one major mailing list—openstack-dev—to discuss the direction of the ecosystem and where the discussions of each independent project take place. There are other mailing lists but are unrelated to software development.

  A former Project Team Leader and Technical Committee member highlighted that cross-project communication occurs in email:

  “Cross-project communication mostly happens over email, which works well.”

  Similarly, a Project Team Leader explained that mailing lists are used for major discussions that require more thought from the participants, as well to reach a broader audience:

  “Use [mailing lists] whenever there seems to be a big discussion with complex thoughts or more formality with a very broad audience, including more informational announcements. Also occasionally for more contentious topics.”

  However, the same Project Team Leader raised some concerns regarding the effectiveness of the mailing lists:

  “I always have some fear of using [mailing lists] because the discussion can sometimes get derailed with no actual decision being made. Sometimes, because there are so many good opinions and sometimes because there are highly opinionated people that speak with authority, yet aren’t really informed enough to provide the opinion. So putting something on the mailing list can have mixed results.”
### Main communication channels

<table>
<thead>
<tr>
<th>Channel</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mailing lists</td>
<td>They are used for discussions about projects. One mailing list stands out as for coordination of the ecosystem: <code>openstack-dev</code>.</td>
</tr>
<tr>
<td>IRC</td>
<td>An interactive Internet text messaging system used in OpenStack as instant messaging for synchronous communication. The Release Team uses it for team discussions and to answer developers’ questions about release management.</td>
</tr>
<tr>
<td>Conferences</td>
<td>General meetings where developers meet in person to discuss and plan the OpenStack development, and to promote OpenStack with users and integrators. A major conference is held every six months.</td>
</tr>
</tbody>
</table>

### Secondary communication channels

<table>
<thead>
<tr>
<th>Channel</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launchpad</td>
<td>A web-based bug tracking system, used by developers and users, to track and discuss bug reports or feature requests. It is also used for lightweight specifications (blueprints) created by the Project Team Leaders to track features for future releases.</td>
</tr>
<tr>
<td>Wiki</td>
<td>A collaborative tool used in OpenStack as the organization space for development as well as the getting started documentation for OpenStack integrators. It originally kept the information of the release process, and instructions for developers to make releases.</td>
</tr>
<tr>
<td>Blogs</td>
<td>Two types: a community blog to inform the ecosystem, and a web-based aggregator of OpenStack developers’ blogs. The Release Team members use blogs to write about the release process.</td>
</tr>
<tr>
<td>Sprints</td>
<td>Focused mid-cycle meetings of developers to accelerate the work on specific tasks towards the next release. There are two kind of sprints: face-to-face and virtual sprints.</td>
</tr>
<tr>
<td>Video-conference</td>
<td>Focused meetings to discuss about a project. Some developers use it as a supplement of IRC, and for one-to-one meetings between Project Team Leaders and community members.</td>
</tr>
<tr>
<td>Gerrit</td>
<td>A website to perform code review and repository management in OpenStack. Every source code contribution must be reviewed in OpenStack. The Release Team uses it to discuss the management of dependencies of OpenStack projects.</td>
</tr>
<tr>
<td>Etherpad</td>
<td>A real-time collaboration tool centred on documents. It is extensively used by Project Team Leaders and Release Team during meetings, specifically at conferences, to keep track of the agenda, decisions and roadmap. The content is later moved to the wiki.</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of communication channels found in the OpenStack ecosystem.

- **IRC.** Unlike OpenStack mailing lists, there are channels whose orientation is specific
for the development of each project. There are also channels to reach multiple stakeholders (users, integrators and developers). Some of these channels help users get answers to questions interactively. There are two important channels for the Release Team: 

#openstack-relmgr-office, used by the Release Team for the team discussions of the release process, and, to answer quick questions from developers about release management; and, 

#openstack-dev, which is a channel the Release Team monitors to be aware of the cross-projects discussions held by the developers. IRC is useful for developers, Project Team Leaders and Release Team to ask quick questions and get immediate feedback, as a Project Team Leader stated as one of the IRC uses:

“I feel that IRC usually works better for quick discussions and decision making.”

However, for deeper discussions, some developers prefer to move a discussion from IRC to a different communication channel: In some cases, IRC is used to reach developers when a topic in the mailing list has not obtained their attention, in particular, for cross-project communication:

“IRC is a fallback for cross-project communication.”

As any synchronous channel, the participation on IRC depends on the location of the developers, and the participation can be challenging if the developers are located in multiple time zones. A Project Team Leader stated:

“The main issue with direct communication (IRC or the like) is that the project is very diverse timezone-wise, so it is hard if not impossible to find time slots that work for everyone. A more asynchronous chat and project management tool would be good I guess (Slack.com or Gittr or the like).”

- **Conferences.** OpenStack holds a major conference every six months, The OpenStack Summit. This event is used to gather in the same place users, supporting organizations, and developers of OpenStack. The OpenStack Summit is composed of three events: (1) the main conference, with talks oriented to users and developers of OpenStack (2) an exposition where companies showcase their products, and (3) a developer summit (the “Design Summit”\(^8\)), to foster the communication and coordination among developers, and discuss the progress towards the next version of

\(^8\)https://wiki.openstack.org/Design_Summit
OpenStack. The conference charges a fee for anyone interested in participating and getting involved. However, to foster participation of active project contributors—the ones who have been active since the previous release, they are exempted of such fee. The OpenStack Summit is a place for cross-project collaboration and an opportunity to “influence” other project’s direction. In words of a Project Team Leader:

“In OpenStack, you can’t ever truly count on anything to “land” (happen) at any given point in time. You can influence and try to make it happen, you can do reviews, you can contribute code, etc. If you really want something to happen from another project that your project is dependent on, you’ll have to ... bring up at the summit as needed.”

The reason is the high bandwidth that a communication channel like face-to-face interactions provide. The bandwidth in a communication channel corresponds to the number of cues in which an information can be transferred [31], such as the speech, tone of voice, inflection, physical gesture among others [31, 34, 33, 133]. A high bandwidth communication channel allows the transmission of more information than a low bandwidth channel in the same period of time. The value of a high bandwidth channel was highlighted by a Technical Committee member, who stated:

“[to collaborate with other projects in OpenStack] we utilize ... high-bandwidth communication such as Google Hangouts, and face to face time at Summits and mid-cycle coding sprints.”

- **Launchpad.** Launchpad⁹ is a Web-based bug tracking system. In OpenStack, Launchpad has two uses: (1) to discuss and track bug reports and feature requests reported by users and developers, and (2) to discuss and track blueprints¹⁰, which are lightweight specifications of feature proposals for core projects and created by their own Project Team Leaders.

The Release Team and developers use Launchpad to keep track of features and critical bugs for future releases. The Release Team uses blueprints to build a short-term roadmap for the ecosystem. Therefore, the blueprints serve as a point of coordination between the projects and the Release Team. As a former Project Team Leader, core reviewer, and Technical Committee member stated:

---

⁹https://bugs.launchpad.net/projectname

¹⁰https://blueprints.launchpad.net/projectname
“[We use Launchpad] to communicate release information between contributors, the Project Team Leader, and release team.”

As a communication channel, Launchpad is limited to blueprints and bug tracking, as a different Project Team Leader stated:

“Good for tracking blueprint and bug status. Nothing else.”

- **Wiki.** According to OpenStack’s getting started documentation\footnote{https://wiki.openstack.org/How_To_Use_The_Wiki}, the wiki is used as “a place to store and discuss requirements, architectural decisions, archive information from meetings and notify about meetings, as well as provide installation, configuration, and getting started documentation for OpenStack integrators.”. Here the Release Team maintains information of the release process, provides instructions for developers to make releases, and provides details about the current release schedule, such as important dates. The Release Team separates the instructions for multiple audiences: everyone, developers, maintainers, and Release Team members. It is also used to keep meetings’ minutes archived for general awareness. Over time, some uses of this communication channel have been replaced by other media; for example, some documentation have been moved from the Wiki to source code repositories because it is easier to maintain, as a Project Team Leader indicated:

  “[We used the wiki for] general design [and] information sharing. This has been used less as people are now putting the documentation into source repositories and publishing it. I’ve gotten to where I don’t really trust the info on wiki to be accurate or up to date.


A similar opinion was shared by a Technical Committee member:

“Used hardly at all, and replaced by in-tree documentation, which is easier to keep up to date and review.”

- **Blogs.** OpenStack has two types of blogs: (1) a community blog\footnote{https://openstack.org/blog/} to inform about the OpenStack activities, and (2) a common location of aggregated blogs of contributors, supporting organizations, and specialized newsletters\footnote{As in GNOME, the common location is called Planet. http://planet.openstack.org}. The community blog
topics include digest of discussions in the OpenStack Developer mailing list, announcements of events and releases, and summaries of any topic to raise awareness of the OpenStack activities. The aggregated blogs contain more technical content than the community blog; in this place, Release Team members write about the release process, which is expected to reach a wider audience than a mailing list, although still technical. The blog posts are also used to remind developers about the roadmap for each release cycle. Contributors also express their points of view regarding the project either in their own blogs or as comments in other developers’ blog posts. Organizations that provide financial support to the OpenStack Foundation can publish their own newsletters in blogs about their contributions to OpenStack.

Blogs, and the newsletters that some blogs contain, are a good source of information for new contributors, as one Project Team Leader stated:

“Early on, I was primarily a consumer of information. This meant I spent hours and days just reading the mailing list, blogs, watching YouTube videos, etc. As I’ve become much more involved in OpenStack as a contributor I rely more on the summit, some OpenStack newsletters and as much as anything learning information from other contributors.”

- **Sprints.** Sprints are mid-cycle events dedicated to work on a specific task towards the next OpenStack release. There are two kind of sprints: face-to-face and virtual sprints. Sprints are organized either to discuss and plan or to get work done. These meetings are organized by developers, and the activities are tracked either in the wiki or in Etherpad. Face-to-face sprints are like small conferences, therefore, they share the same advantages and disadvantages than conferences. Virtual sprints are held on IRC, video-conference, or a combination of both; hence, they share the same advantages and disadvantages than IRC and video-conference.

- **Video-conference.** Some OpenStack contributors use Google Hangouts\(^\text{14}\) for team or project meetings, as a complement of IRC or mailing lists. These meetings are used to accelerate the discussions and decision making process in a team or project. For awareness, after a meeting a participant sends a summary to the openstack-dev mailing list or publish the summary in the wiki.

A Project Team Leader emphasized that this communication channel is an important

\(^{14}\text{https://hangouts.google.com/}\)
supplement of other channels, in spite of its shortcomings and lack of support by other developers:

“[Video-conference] is actually **HIGHLY** effective as a supplement to the other forms, but **VERY** looked down upon by some people in the community. There is a feeling that it isn’t inclusive enough and since it isn’t logged, no way to go back to review the logs. This is sentiment is shared even if the video conference is publicized for all to join. There also are not supported open source tools that everybody seems to be able to use in large groups. Google hangouts has a limit of 10.”

For a different Project Team Leader, it is a good channel for one-to-one meetings:


- **Gerrit.** Gerrit\(^\text{15}\) is the code review system used in the OpenStack ecosystem. Gerrit provides a web interface for performing code review and repository management of software repositories using Git\(^\text{16}\). Every piece of code in OpenStack must be reviewed using Gerrit before entering the main repository. Only OpenStack Foundation members can propose changes, perform review, and approve contributions to the OpenStack source code. Through Gerrit, the Release Team discusses with developers the addition, modification or removal of dependencies of OpenStack projects. These discussions can happen simultaneously in Launchpad and Gerrit. For example, a developer indicated that Gerrit can also be used for tracking specifications instead of Launchpad:

“[To coordinate activities] we use ... specs in a spec Gerrit repository.”

- **Etherpad.** Etherpad\(^\text{17}\) is a real-time collaboration tool centred on documents. Projects can keep multiple Etherpad documents. Sometimes developers leave notes to others on the documents, to thrive further discussion either as a comments on the documents or using other communications channels. In other instances, Etherpad is used to collect and leave relevant links related to a project. A Project Team Leader stated:

\(^{15}\)http://review.openstack.org/
\(^{16}\)https://www.gerritcodereview.com/
\(^{17}\)http://etherpad.openstack.org/
"Great for real time collaboration, note taking, and brainstorming, but nothing particularly long term. [I] usually go to Etherpad for any co-designing, keeping meeting agendas, etc."

We observed that Etherpad is extensively used during the Design Summits, where the developers contribute to the agenda for each session or meeting, keep record of the decisions taken, define roadmaps, and anything that later might be edited to be published on the wiki. A Project Team Leader confirmed this in the interview:

"[We use Etherpad] at summit time for communication of the tracks we run, and to collect information during bug triage."

In OpenStack, the Release Team and Project Team Leaders use mailing list, IRC, and face-to-face interactions at conferences as the main communication channels for coordination. The mailing list is used for long term discussions, IRC for quick discussions, and face-to-face interactions every six months at meetings during the Design Summit. However, the Release Team uses other channels as support for their work, such as Launchpad to keep track and communicate releases, follow blogs as input from developers, video-conferences for one-on-one meetings with Project Team Leaders.

4.4.1.2 How do developers communicate and coordinate for release management?

From our analysis of the openstack-dev mailing list, we found that it is used for multiple purposes. The mailing list is used to discuss the development per individual projects as well as for discussions about the roadmap of the ecosystem. Nevertheless, the openstack-dev mailing list is used by OpenStack contributors, to discuss precise topics, ideas, plans and any subject related with the development of OpenStack. The discussions that belong to individual projects are distinguished of the general ones as they contain one or more topics.

Twelve themes of discussions emerged, six of them are directly related to release management activities. Table 4.2 shows a summary of each theme.

The themes related to release management activities are:

- Request for comments: This is a specific type of proposal. It may be tied to a particular project or have the potential impact on other projects or teams. Developers request comments from peers to nurture ideas that later can become a formal
## Discussions related to release management

<table>
<thead>
<tr>
<th>Theme</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request for comments</td>
<td>Short-term planning, and previous discussion before proposing a long-term plan</td>
</tr>
<tr>
<td>Proposals and discussions</td>
<td>Long-term planning, roadmap discussions</td>
</tr>
<tr>
<td>Announcement</td>
<td>Announce releases and create awareness of the status of each project in the ecosystem</td>
</tr>
<tr>
<td>Reminders</td>
<td>Specific type of announcement, mainly to keep developers on schedule</td>
</tr>
<tr>
<td>Request for decision</td>
<td>A decision is requested from a small group of people</td>
</tr>
</tbody>
</table>

### Other type of discussions

<table>
<thead>
<tr>
<th>Theme</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Events coordination</td>
<td>Organization of conferences, meetings, or related activities</td>
</tr>
<tr>
<td>Expertise seeking</td>
<td>Developers seeking people who work in specific artifacts</td>
</tr>
<tr>
<td>Knowledge seeking</td>
<td>Developer seeking specific knowledge (either technical or organizational)</td>
</tr>
<tr>
<td>Reports</td>
<td>Report of meetings to create awareness</td>
</tr>
<tr>
<td>Out of scope</td>
<td>Anything else</td>
</tr>
</tbody>
</table>

Table 4.2: Summary of categories found, grouped in related and unrelated to release management activities.

Proposal, or to discuss short-term impact of individual projects that may affect the ecosystem. For example, organizations supporting OpenStack usually build and offer custom versions of OpenStack. These organizations may request comments about making some custom features part of the official OpenStack. Thus, before preparing a lightweight specifications or blueprint, these organizations try to obtain the “feeling” of the community, to learn about the developers’ concerns, and to be sure that a blueprint would be accepted before investing time working on them. For example, a developer of an organization asked about interest from other developers before moving forward:

“If anybody is interested in this, we want to contribute for this as [a blueprint] and make it upstream.”

The developers of projects interested in such features might support the idea or raise concerns.

Another example of request for comments is “Separating our Murano PL core in
own package”. Murano is an application catalog that may have grown enough to split it, but before proceeding; the opinion of other developers is requested.

Figure 4.4 shows that requests for comments happen through the whole development cycle.

- **Proposals and discussions:** These messages are long-term proposals that affect the entire ecosystem and require a high level of coordination. In OpenStack there are multiple type of proposals. Some proposal may involve discussing the vision of the project for the next release or major changes whose execution could take one or more releases. Another type of proposal is the request to incubate new projects under the umbrella of OpenStack.
These proposals are usually presented as blueprints. Although blueprints are uploaded in another communication platform (Launchpad), a link to the blueprint is usually submitted to the mailing list to either invite further discussion in Launchpad or to discuss them directly in the mailing list. Sometimes, the proposals are written using Google Docs or Etherpad. Another kind of proposal corresponds to the nomination of contributors for a new role, for example, to become an official member of a team. For example, the incubation of a new project for key management service was proposed with a blueprint and a discussion under “Barbican Incubation Review”. Another example is “Blueprint: list-tested-apis”, which is an email that summarizes a blueprint proposing a mechanism to list the APIs available and find which APIs need unit tests.

Figure 4.5: Proposals and discussions during the release cycle.
Figure 4.5 shows that proposals and discussions happen through the whole cycle with a slight emphasis towards the first and second milestones.

- **Announcement**: These are 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 development branch, incoming major changes in a project, reporting on the status of project goals, candidacies to become a Project Team Leader, and updates in the services infrastructure of the ecosystem.

For example: “Self service StackForge project creation” is the announcement of a new service for continuous integration available for developers; and “taskflow 0.2.0 released” is the announcement of a new version of a library to handle states and failures.

Figure 4.6 shows when announcements occur during the release cycle. They are very few (1 to 3) with some spikes.

- **Reminders**: This is a specific type of announcement that the Release Team uses to send periodic reminders about the current release cycle stage. These messages usually include the outstanding bug reports separated by project and that must be addressed before the release. The Release Team reminds developers the upcoming milestone, release of a new version, and so on. Additionally, it is used to remind the participants of the social norms to be followed during the discussions. Its nature and recurrence make it worth a theme by itself, even though they are a type of announcement. For example, “Review requests on the mailing-list” is a reminder for developers to refrain themselves to request reviews via the mailing list because Gerrit already provides a notification system, and therefore, it is redundant. Another example is “Voting for [Project Team Leader] for Cinder, Heat and Horizon closes in 24 hours”, which is to remind active project contributors of the projects Cinder, Heat, and Horizon to vote for their Project Team Leader before the deadline.

Figure 4.7 shows that reminders occur mainly at the beginning of the cycle and the second milestone. The second milestone involves the release of a development version as well as the start of the feature freeze.

- **Request for decision**: There are two kind of requests for decision: first, to request the breaking of a freeze period starting in the second milestone, and second, to request a prompt decision regarding a pending proposal or, sometimes, when a code
Figure 4.6: Announcements during the release cycle.

review has not reached consensus. The decision to accept or reject a break in the freeze period is taken by the Project Team Leader of each project, although it may be advised by the Release Team. For example: “Wait a minute... I thought we were going to remove Alembic until Icehouse-1?”.

Figure 4.8 shows that requests for decision happen only towards the end of the release cycle.

The remaining five themes are less relevant to release management activities:

- **Events coordination**: Messages to announce, discuss agenda, and coordination events such as weekly meetings, conferences, and sprints. An example of discussion
in this category is “Summit approaching – Submit your session proposals now!”
which is an invitation to send proposals for sessions at the main conference.

- **Expertise seeking:** These are questions from developers who seek others who work
  on or are in charge of specific projects, libraries, or other artifacts within OpenStack.
  A common case of expertise seeking in OpenStack is looking for a developer to
  review code proposed in Gerrit. For example, “Ask for help reviewing nova-specs”
  corresponds to someone looking for an experienced Nova developer to review a patch
  in Gerrit.

- **Knowledge seeking:** These are questions from developers about specific technical
  or organizational issues. A common case in OpenStack is to understand errors
that occur when developers try related projects that are not their own. For example, “OpenStack python clients libraries release process” corresponds to someone asking for clarification of the release process for libraries.

- **Reports**: Messages send by participants in an event who report the discussions and decisions taken during such event. For example, “Meeting minutes and log - 05/26/2014” is a report of the meeting minutes taken for the project Minstral and for awareness of the ecosystem.

- **Out of scope**: In this theme, we collected any other message, such as end-user support requests, invitations to participate in a research survey, and generic complains. An example of an *out of scope* message is “Biggest Fake Conference in Computer
Table 4.3 shows both, the number of discussions and messages during the period studied. Individually, there are less Proposals and discussions than Announcements type of discussions. However, the proportion of messages reflects that Proposals and discussions is an important part of the discussions in the mailing list. Another type of conversation that is prominent the mailing list is knowledge seeking.
4.4.1.3 Who are the key actors in the release management process?

The OpenStack Foundation employs one developer to work on release management\textsuperscript{18}, therefore, it is the main point of contact for anything related to release management: it coordinates with developers, Foundation, or any stakeholder inside the OpenStack community.

The main contributors of an OpenStack project are: the Project Team Leaders, core developers, and active project contributors.

Within each project, the Project Team Leader is in charge of release management; core developers are the ones who make significant contributions over multiple releases; and, active developers contribute regularly with code and review within a single release cycle.

In OpenStack, every six months each project elects a Project Team Leader. In the election process, only the active project contributors can vote. The Project Team Leader has the final word on decisions regarding the project. Internally, the Project Team Leader coordinates the project to set the plans for each release cycle, and it is the liaison between the project and the Release Team.

The Release Team and Project Team Leaders work together on release management tasks, such as the discussion of the roadmap for the next release(s) of OpenStack, and the direction of each individual project. Any person involved in the release management tasks can invite other contributors to participate in these discussions.

As described in Section 4.2, we conducted a social network analysis to identify the participation in the communication process of the developers involved in release management, and to determine if any other contributor would play a key role in the release management process. We also employed the analysis to understand the relationship between the people working in the release management process with the communication and coordination in the discussions of the whole project. In GNOME, this analysis was crucial to determine developers involved in cross-project coordination and release management activities. In OpenStack, this information was already available in the documentation of the project; thus we used this analysis for comparison purposes.

Figure 4.9 shows the social network of openstack-dev mailing list discussions. We selected the six more representatives interactions of Release Team members. The same figure is split into six sub-figures to highlight the interactions of each selected Release Team member (labeled from Release Team #1 to #6, from Figure 4.9a to 4.9f), and how

\textsuperscript{18}When we started our study, the job title was “Release Manager”, which later changed to “Director of Engineering”.
Figure 4.9: Scope of interaction of different Release Team members in the mailing list social network. We selected the six more representatives interactions of Release Team members. 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.

they reach different participants in the social network. The box in each sub-figure 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.
In OpenStack, the release management responsibilities are shared between the Release Team and the Project Team Leaders. The latter are elected every six months by the active contributors of each project, and are the liaison between the projects and the Release Team. Thus, the Release Team obtain insights from every project within the ecosystem.

### 4.4.2 What are the release management tasks in the OpenStack ecosystem?

In OpenStack, the release management tasks are shared between the Release Team and the Project Team Leaders. They are responsible for: (1) defining the requirements of an OpenStack release (2) coordinating and communicating with projects and teams to reach the objectives of each milestone (3) coordinating feature freeze exceptions (FFEs) at the end of a release, and (4) shipping a release within defined quality and time specifications.

To define the requirements of an OpenStack release the Release Team and the Project Team Leaders start a discussion at the Design Summit, which continue later in the `openstack-dev` mailing list; they discuss the specifications and goals of core project towards the next release during the Design Summit and following weeks (as shown in Figure 4.10; for a detailed description of the main milestones, see Section 4.3). Based on the schedule, contributors propose new features in the planning stage of the release cycle. The features are approved by the Project Team Leaders of each project, and the major features are tracked in blueprints. Each project has freedom to discuss and plan the features that will be developed for each release. To track blueprints and coordinate the development efforts, the
Project Team Leaders set a priority and a milestone for each blueprint.

To coordinate the projects and teams to reach the objectives of each milestone, the Release Team and the Project Team Leaders make sure that the development objectives are correctly communicated to the OpenStack ecosystem, by ensuring the cycle and milestone plans are kept up-to-date in the documentation page\textsuperscript{19} of the Release Team. The development is done in branches depending on the specification or blueprint. These branches live in the source code repository. The purpose of the last milestone ($M3$ in Figure 4.10) is to let developers focus on stabilization, packaging and translation; to that end, the source code repositories of the core projects are tagged, and no new features can be added after this point. The Project Team Leaders monitor the release schedule and remind their team members of deadlines.

To coordinate feature freeze exceptions (FFEs) at the end of the release cycle (See Section 4.3), the Release Team helps developers decide when exceptions should be granted. The Project Team Leaders are the ones who evaluate and approve or reject such exceptions; therefore, they monitor blocking bugs and development of features that must be completed before a release.

To ship a release within defined quality and time specifications, the Release Team takes the packages of all individual projects that conform the integrated release, integrates them, and delivers the official milestones and releases of OpenStack. This task involves the integration of every component of the release set and validates that the software runs as expected. The Release Team tracks progress of development teams against published objectives for each milestone, and holds weekly meetings on IRC, all of these activities oriented to ensure the timely publication of each release deliverable. In addition, the Project Team Leaders monitor that the releases-related patches in their project are reviewed in a timely manner

\[\text{The Release Team and Project Team Leaders define the requirements of every OpenStack release, coordinate the projects and teams to follow the release schedule, meet the deadlines and reach their objectives of each milestone, coordinate the feature freeze exceptions at the end of the release cycle, integrate and test all individual packages that conform a release, and release OpenStack.}\]

\textsuperscript{19}It used to be documented in the wiki page of the Release Team, but at the time of this study it had been moved to a repository from where the documentation page is generated automatically. See the description of the wiki in Section 4.4.1.1 from more details.
4.4.3 What are the challenges that release managers face in the OpenStack ecosystem?

As described in section 3.1, we consolidated information found on OpenStack’s websites, and used this information as a guide to conduct semi-structured interviews in which we asked release managers and the Project Team Leaders about the major challenges they faced. We identified the following major challenges: they need to (1) coordinate projects and teams without direct power over them (2) keep everyone informed and engaged (3) decide what becomes part of the integrated release (4) monitor changes (5) set priorities in cross-project coordination, and (6) overcome limitations of the communication infrastructure.

4.4.3.1 Coordinate Projects and Teams Without Direct Power Over Them

Michlmayr [108, 109] defines volunteer as a participant who is free to choose where to work regardless if it contributes in its spare time or employed by an organization. We use the same concept of volunteer, because the Release Team does not have any official power over developers.

One challenge the Release Team faces is to convince developers of its judgment and knowledge in the release process. In any FOSS project with a diverse base of volunteers, many of them will focus on solving their own problems, which may or may not align with the priorities of the each project or the official release:

“We don’t have direct control over contributors’ activities ... In a healthy Open Source project, you have a lot of people contributing to the project, a lot of diverse interest, a lot of people who are just scratching an itch... so it is difficult to focus the people on a set of features that are priority.”

In addition, each project is independent in their decision making process, allowing each Project Team Leader to follow its leadership style: take all the decisions unilaterally, build consensus on every decision, or anything in between both.

OpenStack is not a project whose developers are volunteers working in their free time. In OpenStack, most contributors are paid by companies to work on OpenStack. An employer assigns one or more people to work on a particular feature in a project depending on the employer’s interest. One way to address this challenge is to work towards the shared interests between the multiple companies involved in the development, influencing indirectly
the developers. As a member of the board of directors of OpenStack Foundation stated in a public talk:

“Most contributors of OpenStack are employed by companies that are building products around of OpenStack. A lot of these companies [share] interests, share views where the project should go. If you are employed by a company, that company can set the priorities in terms of the things you spend time on.”\textsuperscript{20}

The Project Team Leader can plan, but the contributors are the ones that implement the features within a release cycle. The Release Team, working together with the Project Team Leaders, can build awareness of their plans and needs with the developers’ project manager at each organization. If the project managers agree with the plans and needs, they can instruct the developers they manage to work on them. Thus, the Release Team can indirectly set the priorities to the developers.

4.4.3.2 Keep Everyone Informed and Engaged

The availability and use of multiple communication channels poses a challenge on how to keep everyone in the ecosystem up-to-date; in particular, because some communication channels are more inclusive than others.

The Design Summit is the major conference in OpenStack. In this conference there are sessions and meetings to plan the release cycle, and it is the main source of major discussions to drive the project forward. However the participation is limited, as one Project Team Leader stated:

“... in OpenStack there are mid cycle meet ups and the every six months summit where only a subset of contributors are able to attend, because companies only have so much travel budget.”

As a consequence, a subset of developers does not participate directly in the discussions, and the challenge is to keep them with the information up-to-date and engaged to meet the goals at the end of the cycle. One way to alleviate the low inclusiveness of the conference, all the sessions notes are written in Etherpad, and some sessions have an IRC channel opened where some developers can provide feedback within the limitations of the channel.

\textsuperscript{20}“OpenStack and Shenanigans – A Case Study in Open Source Governance”. Talk by Mark McLoughlin at Linux.conf.au in January 2014. https://youtu.be/dvmlXDToorM?si=37m
4.4.3.3 Decide What Becomes Part of the Integrated Release

In OpenStack, new projects can request guidance in the release policies in a process of “incubation”. The projects “graduated” from the incubation process can request becoming part of the integrated release. It is in the best interest for new projects to become part of the integrated release, because only these can use the “OpenStack” trademark. Furthermore, this may be an incorrect incentive for new projects: to become part of the integrated releases as soon as possible, and not necessarily when they are ready for production.

As more and more projects become part of OpenStack, the ecosystem grows in scope and poses new challenges to be managed. Adding new projects increases the number of projects to monitor, increases the number of dependencies required to build the integrated release, and increases the demand of cross-project resources and collaboration. However, it also sets expectations among the consumers of OpenStack. This point is particularly important because the integrated releases comprises older and mature projects with new and recently “graduated” projects.

Once a project becomes part of the integrated release, distributors of OpenStack are likely to put them in the users’ hands, hence, are expected to be ready for production. Distributors raise the visibility of the project, which may lead to more deployments, and eventually to attract more contributors. For users (in OpenStack they are also called operators), the integrated release may be interpreted as a set of mature projects ready to be deployed and used in production. Although the Release Team works together with the Technical Committee to resolve when a project is actually ready to be included, new projects may not meet the expectations of some operators. A Project Team Leader emphasized this point:

“Some operators of OpenStack have interpreted the adding a new project to the integrated release as the projects are mature, they [are] ready for production usage, they are ready for large scale usage; and that turns out not to be the case.”

A solution proposed, and later adopted, to address this challenge is to drop the concept of an integrated release and adopt a tagging system for projects, where a project is tagged according to its maturity level. For example, tags for core libraries, tags for projects that have been deployed in large scale, tags for recently “graduated” projects. In summary, tagging is a mechanism to provide extra information of the expected components of OpenStack to consumers, and thus reduce the pressure on the Release Team to approve projects.
4.4.3.4 Monitor Changes

The Project Team Leaders need to monitor changes in the project they depend on in order to avoid regressions, and the Release Team needs to monitor changes that may delay or affect the quality of the integrated release.

In spite of coordination efforts, changes in one project might trigger regressions on a dependant project. Some Project Team Leaders rely on tests as an awareness mechanism of the activities of other projects. Thus, testing of the dependencies of a project helps avoid issues. In words of a Project Team Leader:

\[\text{Once\ other project integrates directly with us; they have tests to tell them when our changes break them.}\]

When a regression happens, the Project Team Leader (or a designed delegate) will follow up with developers of other projects. Depending on the stage of release cycle it will be the mechanism to raise the issue: from filing a bug (in a non-critical stage of the cycle) to direct communication to the developers to raise the priority of the issue (in a critical stage of the cycle). As the code base increases and the projects are maintained by different developers, it becomes harder to keep track of the discussions in all the relevant projects, in particular of unintentional breakages before a release. For developers of a project integrated with eight to ten others within the ecosystem, it is hard to be aware of the status of any other project; usually they become aware of changes that affect them indirectly through test failures:

\[\text{[Our Project] integrates with (ideally) all the other OpenStack projects at the API layer. Right now we integrate with 8-10. We obviously can’t keep on top of all the other projects so we typically learn about pertinent changes to other projects through test failures.}\]

4.4.3.5 Set Priorities in Cross-Project Coordination

Because projects are autonomous in their decision making process, it is challenging for a project to set the priorities of another project that it depends on. When a project requires a new feature from another project, the Project Team Leader must coordinate with the target project, build consensus to accept the feature and add it to the target project’s agenda for the release.
However, having the requirement of one project in the agenda of another project does not guarantee its implementation. One way to address this challenge is to commit a developer to become actively involved in the other project. By doing that, a developer can build technical merit that can facilitate the cross-project coordination, gather consensus of the usefulness of the requested feature and implement it. As a Project Team Leader declared:

“The single most effective way to get your features in and affect a project is to become very highly involved in the project as a whole, providing code reviews, bug fixes, spec reviews, attending IRC meetings, for a sufficient period of time that you are recognized as a serious project contributor. This takes a lot of time and typically comes at a cost of not being able to be highly involved in other projects.”

4.4.3.6 Overcome Limitations of the Communication Infrastructure

The communication channels used in OpenStack have shortcomings that may limit the coordination among teams. One example is Launchpad—the platform used for blueprints—, which is limited to handle blueprints per project and it is not capable of handling blueprints across projects, the same blueprint is posted to each project in Launchpad and somebody is responsible for tracking changes to each copy and synchronize them accordingly. Another example is the use of synchronous communication channels like IRC, which are sensible to the time zone of the participants, and yet the Project Team Leaders are required to be available to answer questions and address issues on IRC, especially towards the end of each release cycle.

The challenges of the Release Team and Project Team Leaders in OpenStack are associated with the size and complexity of managing multiple independent projects, developed by volunteers, in a distributed setting. They need to keep the developers informed and engaged to follow the priorities of the official release.

4.5 Discussion

In this section, we discuss our findings and present some lessons we learned from studying release management in the OpenStack ecosystem. We further discuss our findings in
Chapter 5, which presents a theory that explains our understanding of release management in FOSS ecosystems.

**Organization.** The OpenStack ecosystem is governed by the OpenStack Foundation, which is an association of persons with some common business interest, and whose purpose is to promote such common interest. The Foundation is the supporting organization that oversees the ecosystem administratively and technically; however, it delegates the technical decisions to a specific technical committee.

The most active contributors are left to solve development coordination problems. The release activities are accompanied by the presence of a coordinating authority: the Release Team. The Release Team is empowered by the Foundation to take technical decisions, in particular, to oversee the release management activities. This coordinating authority is in line with the literature on single projects, where a person is in charge of the release activities [37].

**Release Team and Project Team Leaders.** The Release Team must communicate to different stakeholders the status of each part in the ecosystem. The Release Team have to give hints to integrators and distributors of the quality and maturity of each component of the integrated release. The developers of each project elect a Project Team Leader, who is the liaison between the project and the Release Team.

Like in GNOME, the Release Team and Project Team Leaders lack direct power over contributors (see Section 4.4.3.1), for which both the Release Team and Project Team Leaders must combine technical and social skills to build trust among developers. For this purpose, it helps that the Project Team Leaders are elected every six months by the active project contributors, and it is expected that those developers trust the skills the elected leader. As a result of the interaction between Release Team and Project Team Leaders, the Release Team can reach every project within the ecosystem, and make them “feel” involved in the release management tasks.

**Release Schedule.** The OpenStack ecosystems follows a time-based release model, where every six months the ecosystem delivers a new version of the integrated product. The release schedule includes a stage for planning, where developers submit proposals for discussion. Planning appears to be important in the OpenStack ecosystem, because good planning enables a clear definition of tasks, and when there is sufficient information to perform a task, contributors can self-assign such tasks with little coordination [27].

Wright [166] argues that planning is difficult in FOSS projects because of the lack of resource management; volunteers come and go, making the resource levels unknown. As a matter of fact, Wright points that in time-based release model, as the deadlines approach,
release managers must decide to omit features or ship them partially implemented. However, the purpose of a time-based release model is to make the release process predictable. As Rossi [140] argues, release managers must prioritize between release on time, with a certain level of quality, and certain features, but both properties cannot be fulfilled simultaneously unless there were unlimited resources. The OpenStack ecosystem is not different to the aforementioned constraints. Once the stabilization stage starts (see Section 4.3), the Release Team and Project Team leaders work closely to assess the features that can be ready on time, and prioritize those that must be stabilized, leaving the incomplete ones for a future release.

Communication. The mailing list is documented as the main communication channel for communication and coordination across projects. However, we found that only part of the coordination happens there. The Release Team holds weekly meetings with the Project Team Leaders on IRC, and at least twice a year, they have a face-to-face cross-coordination meeting at the Design Summit. To understand the release process, the analysis of the mailing list is necessary but somewhat insufficient. At the end of the release cycle, to make quicker decisions, Release Team and the Project Team Leaders communicate more over IRC.

IRC was acknowledged as a synchronous communication channel useful for quick decisions involving a few number of developers. In the context of release management, IRC is used for holding weekly meetings with the Project Team Leaders and for solving immediate issues at the end of the release cycle. As a synchronous channel, our interviewees stated that one of the issues is the timezone differences between developers.

Video-conference was mentioned several times as a good communication channel because it provides “high bandwidth”, which enables the discussion of a wide number of topics in a short time, that in asynchronous channels would take longer. However, there are three major shortcomings: (1) it is not archived, and if it were it might not be “search-able”, (2) some developers are reluctant to use proprietary solutions and there is no good FOSS alternatives for video-conferencing (3) it is not inclusive because there is an upper limit of ten concurrent participants in Google Hangouts (the technology stated to be used).

Mailing lists are good for broadcasting and reaching a wider audience, but it can be slow and the focus of a given discussions can be diverted.

The documentation of the release process must be updated as the release process is adapted to the needs of the OpenStack ecosystem. Perhaps for that reason, we found abundant information regarding release management, however, sometimes it was outdated or inconsistent. This was confirmed by some of our interviewees, who claimed, for example,
that keeping the wiki updated was challenging, and therefore the information there could be unreliable.

### 4.5.1 Lessons Learned

We summarized our lessons learned from the OpenStack ecosystem as follows:

**Delegation of release management tasks helps reach the whole ecosystem.** The Release Team needs to have first-hand knowledge and understanding of the different projects and teams. The Release Team also needs to build awareness in each one of these projects. For that purpose, the release management tasks and responsibilities are shared between the Release Team and the Project Team Leaders. The Release Team have the big picture of the integrated product, whereas the Project Team Leaders have the inner understanding of each project; the joint work provides a diverse set of points of view. Having a shared responsibility, both the Release Team and Project Team Leaders may feel more affinity to the challenges and problems of each side, especially when each side decisions may contravene the wishes of the other side. Regardless, both the Release Team and the Project Team Leaders are expected to make the best decisions that benefit both the ecosystem and the individual projects as a whole.

**A well defined release process helps keep the process under control.** The developers need to know the release process, with clear stages and milestones, and good understanding of the communication channels to use in each stage to reach the Release Team. The job of the Release Team is to build such schedule with the feedback of the Project Team Leaders, because a clear process and definition of tasks reduce the need for coordination [27] and it helps keep the release cycle under control. When both the Release Team and Project Team Leaders share responsibilities, it is also crucial to have a good understanding of the duties of each one; updated and comprehensive documentation of the release process and responsibilities helps avoid misunderstandings and facilitate the communication flow to developers. In addition, there are planning sessions at the beginning of each cycle at the OpenStack Conference (Design Summit) between the Release Team and Project Team Leaders. As Olson and Olson argue, the collocated activities make easy to provide individual context, to build awareness of what the participants have in common, and therefore, such activities may alleviate the communication needs and may increase the usefulness of other communication channels (like email and IRC).
The Release Team must be willing to redefine the official release. The concept of “official release” in OpenStack lead to wrong assumptions in the OpenStack ecosystem; distributors and integrators assumed incorrectly that all of these projects shared the same level of quality and maturity. If the Release Team assesses that the concept of “official release” sends the wrong message to distributors and integrators, then it must be willing to adapt the concept or drop it. For example, some projects in the “official release” were five years old, with a record of deployments, whereas new projects were not equally stable nor mature, and yet, they became part of the “official release”. As a consequence, the Release Team decided to adjust the rules and replace the concept of “official release” by a tagging system. At the end of each release cycle, the Release Team tags the applications of those projects that follow the release schedule\textsuperscript{21}, and the Release Team does not make any ”official release”. The applications tagged according to adoption (number of deployments running the application), maturity (level of stability), and age (number of years since a project started its development). This avoids misleading distributors and integrators regarding the expected quality and stability of every project in the official release.

Need a common communication channel for coordination. We observed that for high level coordination the Release Team uses a mailing list. If any important discussion occurs in a different communication channel (for example, during a meeting on a conference or IRC), a participant of such discussion is expected to write a report and send it over the mailing list, for both, awareness and further discussion. Such good practices have also been reported by Mockus et al. [116] on regular development settings. A mailing list is not a rich media channel like face-to-face interactions or video-conferences, but it provides several benefits that are valuable within FOSS ecosystems: they are archived, searchable, and asynchronous. They serve as a “database of wisdom” that developers can search for past discussions, and they allow the participation in the discussions of developers geographically distributed.

4.6 Threats to Validity

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

\textsuperscript{21}Some projects may decide to not follow the release schedule, and therefore, out of the tagging system.
4.6.1 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 study relied on data publicly available, specifically on the public archives of the mailing list we found. We identified three primary communication channels in OpenStack that are archived: mailing lists, IRC, and face-to-face interactions at conferences (Design Summit). We also found complementary communication channels that place an important role next to the primary ones, such as video-conference. Most of the discussions held in mailing lists are concentrated in one single mailing list: openstack-dev. This mailing list serves two audiences, those interested in (1) project specific discussions, and (2) general discussions of the whole ecosystem. Our study was focused on the latter, and therefore, we filtered the messages to analyze only the general discussions. To minimize the possibility of missing relevant interactions, we applied the same filtering criteria that is suggested to subscribers of the mailing list. Regarding the selection of a mailing list instead of other communication channels, we relied on previous research that suggest the majority of discussions occur in mailing lists [41, 43, 13, 116, 117], and to make this case study comparable to our study on the GNOME ecosystem. To narrow the possibility of missing important coordination activities, we conducted interviews to triangulate our findings with key developers we identified in our study.

4.6.2 Internal Validity

Internal validity relates to the validity of causal inferences made in the study. Researcher bias is a threat to the internal validity. The manual coding of emails relies on human judgment and I might have introduced bias in the results. I abstracted common patterns in the communication process by following Creswell’s guidelines [25] for coding; I segmented sentences into themes and label them with a term. I interpreted the subject field and content of each email thread to extract the topics, and build the themes. As a consequence, it is possible I misinterpreted the actual discussions. To alleviate this issue, before coding I familiarized myself with the discussions and, afterwards, I triangulated the results by interviewing developers.

To identify key developers to interview, I explored OpenStack documentation and applied social network analysis on the openstack-dev mailing list. Because of the limited information available in the mailing list archives, the resulting social network was not as rich as in the GNOME case study. However, the documentation about roles was compre-
hensive; I extracted from that documentation the Project Team Leaders and Release Team members, to whom I extended invitations to participate in this study. I used the social network analysis to verify the participation of the candidates in the discussions, but not as the primary metric to determine key developers.

### 4.6.3 External Validity

External validity is concerned with the extent to which it is possible to generalize the findings. A single case study may impose a threat to generalization of the results. However, this is the second case study performed and the comparison with the GNOME case study is addressed in Chapter 5.

### 4.6.4 Reliability

Reliability refers to the research steps, such as the data gathering, can be replicated and lead to similar results. The mailing list data is publicly available, and our procedure is clearly described. The procedure includes the selection criteria of the FOSS ecosystem to study, the selection of key contributors to interview, the interview questions, and how we performed the analysis of the data using standard practices as described by Creswell [25], and Corbin and Strauss [24]. Other researchers can replicate the steps of our research, and we expect that such replications lead to similar results.

### 4.7 Summary

In this chapter, we explored the communication channels used in the OpenStack ecosystem for communication and coordination towards the release of an integrated release, we also explored the release management tasks and the challenges that release managers face in the ecosystem.

We determined the communication channels used for cross-project coordination within the OpenStack ecosystem, and how they used each one of them. The OpenStack’s developers communicate and coordinate using multiple communication channels, both asynchronous (email, blogs, launchpad, wiki, gerrit) and synchronous (face-to-face interaction in conferences and sprints, IRC, video-conference, etherpad). We identified mailing lists (openstack-dev, IRC and face-to-face interactions at conferences as the main communication channels. Mailing lists for asynchronous discussions, IRC for synchronous communi-
cation and to answer developers’ questions about release management, and conferences to discuss and plan the OpenStack development.

We further studied the *openstack-dev* mailing list, because it comprises the discussions of the OpenStack ecosystem, it is archived, and to make this case study comparable against the GNOME ecosystem presented in Chapter 3. When we characterized their communication patterns; we found that developers mainly use the mailing list to nurture ideas (*request for comments*), and discuss formal proposals (*proposal and discussions*).

We also identified the central actors in the communication regarding release management, and we conducted interviews with a subset of these relevant actors, from whom we obtained further insights and triangulated our results. In OpenStack, we identified that Project Team Leaders and the Release Team members are the key players in the communication and coordination across projects in the context of release management. The communication is spread across several communication channels, depending on the stage of the release schedule. However, we established that the main communication channel is the *openstack-dev* mailing list, with an increase relevance of IRC towards the end of the release cycle. Also, face-to-face meetings are relevant at the beginning of each release cycle.

In OpenStack, the Project Team Leaders are elected among the active project contributors within each project every release cycle, and they are the liaison between each project and the Release Team. The role of the Project Team Leaders is clearly defined, who also share responsibilities in the release process with the Release Team. Because the Project Team Leaders are elected by their peers in each project, the Release Team can rely that any decision taken with the participation of the Project Team Leaders will be transmitted and executed accordingly within each project.

We also determined the release management tasks are shared between the Release Team and Project Team Leaders. These tasks are: (1) defining the requirements of an OpenStack release (2) coordinating and communicating with projects and teams to reach the objectives of each milestone (3) coordinating feature freeze exceptions (FFE) at the end of a release, and (4) shipping a release within defined quality and time specifications. Thus, the Release Team and Project Team Leaders define the requirements of every OpenStack release, coordinate the projects and teams to follow the release schedule, meet the deadlines and reach their objectives of each milestone, coordinate the feature freeze exceptions at the end of the release cycle, integrate and test all individual packages that conform a release, and release OpenStack. A clear definition of roles and responsibilities facilitate the communication flow and coordination across projects towards a common effort: an integrated release.
We also identified the challenges faced by the Release Team and Project Team Leaders in the OpenStack ecosystem. The six major challenges we identified in this study are: (1) to coordinate projects and teams without direct power over them (2) to keep everyone informed and engaged (3) to decide what becomes part of the integrated release (4) to monitor changes (5) set priorities in cross-project coordination, and (6) to overcome limitations of the communication infrastructure. Like in the GNOME ecosystem, the challenges are related with the size and complexity of managing multiple independent projects, developed by volunteers, in a distributed setting. The Release Team and Project Team Leaders need to keep the developers informed and engaged them to follow the priorities of the official release of OpenStack.

The contributions of this chapter are: an empirical study of release management in a FOSS ecosystem, and a set of lessons extracted from this case study. These contributions are important to build and refine a theory of release management discussed in Chapter 5, which we compared against the GNOME ecosystem presented in Chapter 3. The lessons learned from this case study are summarized as follows: in the OpenStack ecosystem (1) the delegation of release management tasks helps reach the whole ecosystem, (2) a well defined release process helps keep the process under control, (3) the Release Team must be willing to redefine the official release, and (4) the ecosystem needs a common communication channel for coordination.
Chapter 5
Discussion and Theory

“A mathematician usually discovers a theorem by an effort of intuition; the conclusion strikes him as plausible, and he sets to work to manufacture a proof.”

—Geoffrey H. Hardy

In this chapter, we present and discuss a theory that encapsulates our understanding of the communication and coordination regarding release management in the ecosystems we studied.

Our research questions addressed the release management in FOSS ecosystems in three aspects: (1) the use of communication channels for communication and coordination towards the release of an integrated product based on multiple projects (2) the release management tasks in FOSS ecosystems, and (3) the challenges that release managers face. To answer these research questions, we conducted case studies in two FOSS ecosystems: GNOME and OpenStack. We presented the results and lessons learned in Chapters 3 and 4, respectively.

Our results show that a Release Team:

• requires both, good technical and social skills
• needs a common communication channel for coordination
• needs members with a variety of backgrounds
• must follow the main communication channels used by developers, and

• defines a clear schedule to facilitate the coordination process.

Based on these findings we developed a theory that captures the essential elements of the release management process, challenges and problems in a way that may be generally applicable to FOSS ecosystems. This theory is formulated as three high-level statements that encapsulate our understanding of the communication and coordination in FOSS ecosystems in the context of release management. This theory can be summarized as:

1. The size and complexity of the integrated product is constrained by the release managers capacity

2. The release management should reach the whole ecosystem, and

3. The release managers need social and technical skills.

The chapter is divided in four parts. The first presents the research framework we used to build our theory. The second explains each statement of our theory of release management in FOSS ecosystems. The third discusses and explores the implications of our theory. The fourth summarizes this chapter.

5.1 Building an Empirical Theory

The term theory has different meanings depending on the field. In computer science, a theory conveys rigorous means of reasoning mathematically from axioms, which usually do not require empirical evidence to be accepted; “they are proved logically, not contingently” [61]. In social sciences, and sometimes in physics, theories cannot be expressed mathematically; what matters is whether the theory accurately describes the phenomena studied. Herbsleb and Mockus argue that an empirical theory in the field of software engineering overlaps several fields in social sciences: anthropology, sociology, organizational science, and psychology [61]. Our theory is in line with this definition.

The development of theories using qualitative research, and in particular case studies, has a long tradition in the social sciences. In software engineering, qualitative research was not widely used by the mid of 2000’s, with a low number of empirical studies [141], which had started after the seminal work of Seaman [147]. According to Seaman [147], the software engineering research community has a pragmatic and result-oriented view on research methodology, rather than a philosophical stand. “The community does not pay any larger attention to the inherent conflict between the positivistic foundation for experiments and the interpretive foundation for case studies” [141].
Exploratory case studies can provide an excellent background from which to develop new theories [171]. One point of contention often raised is whether the findings from a case study are generalizable. While it is not possible to treat a case study as a sample from which to draw conclusions about a bigger population (in the manner of a statistical generalization), case studies allow analytic generalization [171]. Case studies can be generalizable to theoretical propositions [171] using a bottom–up approach: the specifics of data produce the generalization of the theory [36].

Glass [46] contends that the practice of computing and software starts first, then follows the theory to formalize such practice; a theory that will evolve and surpass the practice at some point in time. In his essay, he enumerates several examples to validate his point; one of them is about time-shared systems: “to understand [time-shared systems], the systems had to be constructed, and their behaviour observed.”

In the early stages of a discipline (like release management), theory can progress by studying practice better; in particular to extract the best of practice. In this work, we explored and derived lessons on FOSS release managements from the actual practice in FOSS ecosystems. We followed the spirit of McLean [102], who argued that “the proper place to study elephants is in the jungle, not the zoo”. Paraphrasing Michlmayr [109], by discovering the essential good practices of release management in FOSS ecosystems, these good practices can be transferred later to other ecosystems.

The development of a theory using empirical research may be inductive or deductive. In the inductive approach, the theory is induced from observations made by the researcher with an open mind, looking for patterns, that later can be related to an existing theory or lead to a new one. In the deductive approach, the research starts with an existing theory, the researcher sets hypothesis which later are either confirmed or rejected. Studies with exploratory characteristics tend to be inductive, whereas those with explanatory characteristics tend to be deductive [141].

Although we could uncover all kinds of relationships in our hard data, it is through the soft data that we can explain such relationships. The gathering of insights about causal links, motivations, and reasons why things happen, enables the researcher to build theories that later can be verified by more objective techniques: “systematic data create the foundation for our theories, it is the anecdotal data that enable us to do the building” [115]. Qualitative data is good for describing, understanding, and explaining. Its focus is in natural settings (real life), it is holistic, and it is a powerful tool to study processes when the data is collected over a sustained period. In this work, we studied more than two years and half of communication in two FOSS ecosystems, using an inductive approach with fo-
cus on holistic understanding of practices and challenges in release management in FOSS ecosystems.

As researchers, we must be aware that there is a risk of building a theory that is narrow and idiosyncratic; although we should aim to build a general theory, we must also be humble and accept that a theory narrowed in scope still is valuable [36]. When building a theory, it is important to explicitly describe the boundary conditions on which a theory applies because it avoids the formulation of theories that seem general in ways that are not useful [57].

By performing two case studies we gathered evidence to increase our understanding of the communication and coordination regarding release management in FOSS ecosystems. As an empirical theory we do not claim that our results are final or definitive because such theories can usually be adjusted in the light of new evidence, and a phenomenon may be explained by more than one theory [61].

Regarding generalization, Yin [171] argues that case studies are generalizable to theoretical propositions. Walsham [164] extends Yin’s view, and specifies four types of generalization from interpretative case studies: (1) the development of concepts (2) the generation of theory (3) the drawing of specific implications in particular domains of actions, and (4) the contribution of rich insights captured from the cases studies that are none of the above.

The research herein presented has led to the creation of generalizations in the last three categories: (1) we used the data gathered in our cases studies to generate a theory of release management in FOSS ecosystem (2) we drew a implications that follow from the theory that are applicable to FOSS ecosystems whose governance is driven by Foundations, and (3) we obtained rich insights from key participants in the release management process from whom we learned a set lessons in the form of tasks, challenges, and understanding of the release management process in FOSS ecosystems.

5.2 Theory of Release Management in FOSS Ecosystems

In this section, we dissect the statements of our theory. These statements capture the essential elements of release management in FOSS ecosystems in a way that may be generally applicable to FOSS ecosystems ran by Foundations:

1. The size and complexity of the integrated product is constrained by the release managers capacity,
2. Release management should reach the whole ecosystem to increase awareness and participation, and

3. The release managers need social and technical skills.

5.2.1 The Size and Complexity of the Ecosystem’s Integrated Product is Constrained by the Release Managers Capacity

Parnas [129] argues that one of the benefits of modularity is managerial: development time should be shortened because separate groups would work on each module with little need for communication. A corollary of Parnas’ argument, applied to our research, is that for a fixed period of time there is a limit of managerial capacity to coordinate an integrated product comprised of multiple modules. As we described in Sections 3.3.2 and 4.4.2, the release managers coordinate the effort to define the integrated product in an ecosystem. This is a crucial task because the product delivered also expresses the identity of the ecosystem: a FOSS ecosystem is primarily known by the product it delivers.

Wright [166] theorized that there is a “relationship between architecture and process complexity”. He argues that the more monolithic an architecture is, the more it will induce a complex release process. And vice versa, the more modular and loosely-coupled an architecture is, the more it will facilitate an incremental release process. Finally, modular architectures may still have cross-component dependencies that necessitate an all-in-one release process. Although this theory was not elaborated in the context of FOSS ecosystems, it can also help explain FOSS ecosystems because the official release is comprised by loosely-couple modules (multiple independent projects) that have cross-component dependencies and work in coordination towards an integrated product. Hence, the highly modular nature of a FOSS ecosystem facilitates the release management activities to release a bigger integrated product.

To handle the release process in an ecosystem, the release managers coordinate the definition of what the integrated product is, and define the release schedule accordingly. The integrated product is the result of the coordination of multiple loosely-connected projects working towards a common goal. The release managers coordinate which projects belong to the integrated product, and which external and internal dependencies are needed for the release.

Our results show that release managers try to keep the number of dependencies as low as possible; nonetheless that as the ecosystem evolves, the components and dependencies tend to increase. At regular intervals, the release managers work to re-define the integrated
product and re-evaluate the dependencies in order to reduce them and keep the process under control.

Aranda [2] argues that the formalization of a process in a software project helps reduce the overall complexity, because once formalized, the process 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. Paraphrasing Aranda [2], 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 [2] is reflected in the way the communication flows in each ecosystem; that is, the patterns or types of discussions held in the main communication channel. For example, we identified themes such as *request for comments* and *proposals and discussions* that are associated to planning activities within the ecosystem. Other themes we extracted, such as *reminders* and *announcements* are associated to synchronization points between projects within the ecosystem.

Finally, we noticed that the message subjects are more structured in GNOME, in a way that makes the topic of each email more predictable. We speculate that this behaviour occur because the GNOME ecosystem is older than the OpenStack ecosystem (it is more than twice older at the time of this study). We based our speculation in that over time developers find a common ground to use more effectively the communication channels used for coordination [126].

### 5.2.2 Release Management Should Reach the Whole Ecosystem to Increase Awareness and Participation

The sense of lack of participation or consideration in the discussions and decision making process may alienate those who cannot participate, and it may erode the sense of community. Furthermore, in a study of burnout and engagement in a workplace, Maslach and Leiter [101] suggest that people favouritism or unjustified inequities tend to developer burnout (or disengagement) over time.

To increase awareness and participation, the communication and coordination of the release process should be capable of reaching as many stakeholders as possible. However, this does not mean that every communication or coordination action should be addressed
to every stakeholder, because they could become overwhelmed by the flow of data. The release managers should focus in (a) communicating properly their message, and (b) making sure the message is pertinent to the goals of the release process. This is in line with the theory of shared understanding of software organizations proposed by Aranda [2]. According to that theory, a message is communicated properly when the message is passed and interpreted correctly, and the communication should provide information that is both relevant and sufficient for the accomplishment of the organization goals.

In other words, it is not enough to send a message. The sender have to do it in a way that is interpreted correctly by the receiver; otherwise there is no communication. For release managers, it is crucial to understand how to communicate correctly. When the message is far from trivial, the communication is prone to breakdowns; it depends on multiple factors such as context, cultural background, language, among others. Hence, to properly reach the ecosystem, the release managers need to:

1. Choose the right communication channels for coordination, and
2. Reach the right people.

5.2.2.1 Choose the Right Communication Channel for Coordination

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 [31]. The cues can be verbal (speech, writing) or non-verbal (seeing, touching, tone of voice, vocal inflection, physical gesture, smelling, touching) [31, 34, 33, 133].

In our case studies, the 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 (for example, some developers cannot attend to a conference, other participants may not be as fluent as English-native speakers).

The archival of discussions could help mitigate the disadvantages of synchronous communication channels. In fact, in our studies, developers acknowledged that archiving a
communication channel is a desirable feature because it enables transparency, which is good for the health of the ecosystem. The archival of synchronous communication channels may avoid communication breakdowns and enhance coordination [30].

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 ecosystems we studied.

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.

5.2.2.2 Reach the Right People

Achieving agreement between participants may be more difficult in electronic media channels than face-to-face interactions [170]. Therefore, virtual teams rely on trust; but it takes time to establish trust in complex environments [65, 87, 108] 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 [59, 60].

To build trust, raise awareness and keep the ecosystem aligned to the goals of the release process, the release managers should 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.

We identified two strategies used by the Release Team to reach the ecosystem: (1) engage the project leaders of each sub-project to accept co-responsibility of the release process, and (2) recruit key contributors from different teams into the Release Team.

Regarding OpenStack, the way the Release Team can reach the whole ecosystem is through the Project Team Leaders, who are the leaders of each project within the ecosystem, and each one is elected by the active developers of their corresponding project. Therefore,
the Project Team Leaders contributors who have direct contact with the developers of each project, who likely trust their judgement. Because the Release Team and the Project Team Leaders share responsibilities in the release process, both groups act together as release managers.

Regarding GNOME, the Release Team tries to get similar results by recruiting its members from multiple areas within the ecosystem. Unlike OpenStack, 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.

5.2.3 The Release Managers Need Social and Technical Skills

In GNOME, the release managers are the Release Team members, whereas in OpenStack are the Release Team members and the Project Team Leaders.

Wright [166, 168] researched the causes of release engineering failures in proprietary software settings, and reported the dominant cause of release engineering failure was social. In particular, the release managers who are organizationally disjoint from the team of developers tend to have more friction in their communication, increasing the potential failures in the release process.

We observed that in FOSS ecosystems release managers require both technical and social skills. Technical skills are important to gain respect from developers and to convince developers of their judgement, and social skills to convince developers to perform actions, especially because of lack of direct power over them. For example, the release managers in GNOME are continuously monitoring the main communication channels; if a contributor stands out for its skills and interest in the release process, the release managers may invite that contributor to become more involved, and eventually to become a release manager. In addition, when the release managers need to recruit a new member, they assess the skills the group of release managers may be lacking or that they would like to develop; then they look for a contributor that might have them.

5.3 Discussion and Implications of the Theory

In this section, we discuss our theory in the light of the results of both case studies, the related literature and the implications of the theory. We compare the results from our case studies of the GNOME and OpenStack ecosystems, what we learned, what challenges they face, and how they face them.
Capacity constraints

The size and complexity of the integrated product is constrained by the release managers' capacity. The release managers need to reduce the complexity of the integrated product in order to allow it to grow.

To keep the complexity under control, the release managers can reduce the number of dependencies, increase the modularity in the ecosystem, and define clearly the tasks, procedures, and schedule. All of them may help reduce the complexity and, therefore, alleviate the coordination and communication needs.

Influence in the ecosystem

The sense of lack of participation or consideration in the discussions and decision-making process may alienate those who cannot participate, and it may erode the sense of community. Hence, the release management should reach the whole ecosystem.

The release managers need to (1) choose the right communication channels for coordination, one that enables transparency and participation (for example, archived, asynchronous, public), and (2) reach the right people, the ones who are trustworthy and can help the release managers to far reach the ecosystem.

Social and technical skills

The release managers need social and technical skills. Technical skills are important to gain respect from developers and to convince developers of their judgement, and social skills to convince developers to perform actions, especially because of lack of direct power over them.

<table>
<thead>
<tr>
<th>Table 5.1: Key concepts for release managers according to our theory of release management in FOSS ecosystems.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity constraints</td>
</tr>
<tr>
<td>Influence in the ecosystem</td>
</tr>
<tr>
<td>Social and technical skills</td>
</tr>
</tbody>
</table>

5.3.1 Evolution in the Use of a Communication Channels for Release Management

In GNOME, we found the stages in the release cycle were reflected in the discussions in the desktop-devel-list mailing list. According to the discussions we studied in the GNOME ecosystem, the messages tended to follow a common pattern at every release cycle. For example, request for comments have a predominance at the beginning of the cycle, request for approvals at the end, announcements evenly during the cycle. In the OpenStack mailing list (openstack-dev), we also found that the messages were in line with the release cycle,
although it was not as evident or clear as in GNOME. To illustrate the difference: when we performed the coding of the discussions, the subjects in *desktop-devel-list* mailing list ( GNOME) were more predictable than in *openstack-dev* mailing list (OpenStack), where we rarely relied on the subjects to determine the topic of the discussion. This may be a reflection of different cultures, or perhaps that the GNOME ecosystem is an older FOSS ecosystem than OpenStack (19 and 6 years old, respectively), and therefore, some mechanisms of communication have become more formalized in GNOME. As we discussed in Section 5.2.1, the formalization of the communication is a mechanism to cope with the organizational complexity [2].

According to Dennis and Valacich [34], the development of standards and norms increases with the experience that a group gain, and as result there may also experience an improvement in the interplay and the tasks they perform over time. The channel expansion theory proposed by Carlson and Zmud [21] 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.

That said, in OpenStack there were two discussion themes that stood out: developers looking for a peer to review patches on Gerrit, and contributors looking for insights after having problems to test an application. We found similar discussion themes in GNOME, but they were less frequent.

We found that cross-posting the same topic in multiple communications channels without adding new information is a bad communication pattern, especially in the OpenStack ecosystem, where we found several messages from developers asking to stop such practice.

Although we found the messages in GNOME more structured than in OpenStack, the perception of two of the developers we interviewed was that the *desktop-devel-list* mailing list was noisy, particularly by endless discussions. What we found were some long and heated discussions, with a high number of messages. But in term of number of discussions, those were minimal. This may be an issue if in the long term some developers refrain themselves of participating in such discussions, and the results of this research may help understand the behaviour in the mailing list better. This anecdotal evidence should not be used in favour or against the general usefulness of the mailing lists; the perception of usefulness of a communication channel changes over time as the communication on such channel evolves [34, 63].

Nevertheless, in both ecosystems the release managers recommend the developers to follow the designated mailing lists (the ones we studied). However, in OpenStack the
recommendation is also stated as a requisite for projects looking to become part of the ecosystem.

Guzzi et al. [54] argue that a mailing list is one of several communication channels used in FOSS projects. We concur, as we noticed in both case studies that developers communicate using IRC, face-to-face interactions at conferences, sprints and hackfests, issue trackers (launchpad and bugzilla), among others. Furthermore, Guzzi et al. [54] 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 the mailing list had changed. For example, both Brooks [18] and Parnas [129] argue that when the tasks are clearly divided, there is less coordination required; and according to Michlmayr [113] a decomposed system allows the division of labour with limited communication through well defined channels. Nonetheless, Michlmayr and Fitzgerald [109] 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 (for example, GNOME) or a specific topic in the same mailing list (for example, OpenStack), and they still need a common channel(s) for cross-project coordination.

In contrast to Guzzi et al. [54], we state that the use of mailing lists depends on the project dynamics, because mailing lists allow 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 [126].

In summary, in both case studies, we found that the main communication channel used for cross-project coordination is a mailing list.

5.3.2 Communication Channels Used for Coordination

We also found that some contributors have strong preferences in the selection of the communication channels to interact with other developers. Important decisions that have implications in the ecosystem tend to be reported in the main channel (mailing list), and continue the discussion there. However, the implication of these preference is that the release managers should be ready to monitor other channels used by developers, at least to gain awareness and to be there to answer questions related to the release process that may arise spontaneously.

The reasons reported by developers regarding their preferences were related to the prop-
erties of the communication channels, and the value perceived that each channel added to their interactions with other developers. We found a clear distinction between two groups of developers with opposite preferences: a group that valued effectiveness and immediacy (synchronous) and a group that valued coverage and archival (asynchronous).

According to the interviews, effectiveness and immediacy allow developers to take decisions faster, obtain quicker feedback, and avoid the perils of the asynchronous channels such as discussions that deviate from the topic. Coverage and archival allow developers to obtain feedback from contributors regardless of their timezone, and provide a “database of wisdom” from which to retrieve the discussions behind the decisions. In between those two groups, we found a third group of developers who acknowledged the benefits and shortcomings of each communication channel, but used all or some of them according to the circumstances.

To illustrate the preferences of communication channels among developers, we discuss the preferences of two synchronous channels: video-conferencing and face-to-face interactions. However, the use of video-conferencing is contentious in parts of both ecosystems, whereas face-to-face interaction is not.

Some developers in OpenStack disregard the use of video-conference technology such as Google Hangouts because it is non-inclusive, as there is a limit of ten people participating in a meeting. Some developers in GNOME disregard video-conferencing for philosophical reasons, specifically if the technology to use (for example, Google Hangouts) is not open, either as a standard, or the unavailability of a FOSS implementation; for them, a non-FOSS application or technology is non-inclusive.

In contrast, face-to-face interactions at conferences 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 do they might not be present during a particular discussion.

Face-to-face interactions at conferences and sprints 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 [138].

Both video-conferences and meetings at conferences need archiving. It is granted they are non-inclusive communication channels by nature, but at least archiving may be a mechanism to alleviate the non-inclusiveness of those developers who cannot participate in them.

To an extent, both communication channels are valuable for developers in both ecosys-
tems. These developers use these meetings to accelerate the discussion and decision making of their projects. For awareness, it is important to communicate to the ecosystem the outcome of a video-conference or face-to-face interaction, or meeting.

Regardless of having communication channels that provide higher bandwidth, an important factor documented by Olsen and Olsen [126] 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 [126]. They also argue that when people have established little common ground, even if they have met in person or have used 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, 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.

5.3.3 Governance and Release Management

Markus [99] defines FOSS governance as “the means of achieving the direction, control, and coordination of wholly or partially autonomous individuals and organizations on behalf of an [FOSS] development project to which they jointly contribute”. According to Berkus [10] there are five types of governance style in FOSS projects: solo, monarchist, community, corporate, and foundation (described in detail Section 2.3.3).

Although both OpenStack and GNOME fit in the Foundation category, they have different governance. Specifically, one major difference between both case studies relies on the difference of power given to volunteers and external organizations; GNOME has limited the direct influence of companies, whereas OpenStack embraces the influence of companies. The concept of Foundation in United States (where both Foundations are based) encompasses different types. The GNOME Foundation is a charitable organization regulated by the Section 501(c)(3) of the Internal Revenue Code, oriented by its activity towards the public benefit. The OpenStack Foundation is a business league regulated by (501(c)(6) of the Internal Revenue Code, which is an association of persons with some common busi-
ness interest, and whose purpose is to promote such common interest. Hence, GNOME’s main goal is public interest, while OpenStack is business interest.

Both Foundations oversee the release process as part of their duties described in their bylaws. However, both Foundations delegate the technical decisions to a specific technical committee. By delegating the technical duties, the board of directors separate the administrative role from the technical ones, this allows different people with different skill sets to work where they can serve best. Nevertheless, the Foundation can take back the control of the technical duties if needed.

Since the beginning of the GNOME Foundation, the GNOME ecosystem limited the influence of external organizations in the decisions within the ecosystem. For that purpose, there were set a set of restrictions to conform the GNOME Foundation: (1) only individuals can conform the Foundation, and they do it as a volunteers, not as representatives of an external organization (2) the board of directors is the liaison between external organizations, represented in the advisory board (3) the board of directors is composed by contributors who act as volunteers regardless of their affiliation, and (4) the board of directors cannot have more than two members with the same affiliation. Thus, the external organizations play an advisory role for the Foundation, and their impact in the decisions of GNOME are indirect at the organizational level, if any.

In contrast, the OpenStack ecosystem does not limit the influence of external organizations. In the OpenStack Foundation: (1) external organizations and individuals conform the Foundation (2) the board of directors is composed by members elected by individuals and members designated directly by top tier external organizations, and (3) the involvement of external organizations in the decision process is determined by their level of financial support (for example, to designate a board member, propose an individual as board member to be voted among organizations of the same level of financial support). As a consequence, the external organizations play an active role in the decisions within the ecosystem.

In GNOME, the volunteers play a major role in the direction of the project, relegating the external organizations to an advisory role. In OpenStack, both volunteers and external organizations play a role in the direction of the project, and furthermore, the level of financial support given by an external organization determines its power in the decision making process. Nevertheless, in both ecosystems external organizations can influence indirectly the direction of the project by paying contributors to work on specific areas within the ecosystem.

In the OpenStack ecosystem, every project has one Project Team Leader, which is elected every six months by the active project contributors (also known as APC) during
the last release cycle. This is what they call “representative governance”, people who participate are the ones that take the decisions. In addition, to foster the participation of active developers, they are exempt from fees in the OpenStack conference. Thus, renewing a project maintainer (leader) is part of the process and it can change relatively fast, reducing the possibility of stagnation for lack of participation of a maintainer.

Among GNOME developers, there is a similar “mantra” regarding who participates in the decision making process, but it is informal. However, in the GNOME ecosystem, each project may have several maintainers (leaders), some of them may even become inactive and still listed as a maintainer unless they explicitly step back (sort of “honorific maintainer” title, perhaps kept by courtesy or in gratitude for their past contributions).

5.3.4 Organizational Structure and Communication Flow

We have seen that the communication flow is flat in GNOME, the communication and coordination is driven by the release managers; every developer participates in the discussions and the maintainers of each project take the decisions. The release managers 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. [108, 110] 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 [110], release managers can proactively omit or postpone unfinished features.

In GNOME there is no formal description of the role of the maintainers, developers, and how they interact with the release managers. The structure has evolved organically along with the ecosystem.

In contrast, in OpenStack it is formally stated that every project must have a Project Team Leader, and that the Project Team Leader must be elected every six months by the active project contributors (the incumbent Project Team Leader may run for re-election). An active project contributor is a contributor with active participation in the last six months (it has submitted code or has performed reviews). The release tasks are shared between the Release Team and the Project Team Leaders. Consequently, the communication and coordination seem more hierarchical than in GNOME because this organizational structure has been defined from the top of the Foundation,
The roles in the decision making process of the Project Team Leaders in OpenStack and maintainers in GNOME differ at the end of each release cycle, when the stabilization phase begins (see Sections 3.2 and 4.3 for details). This period is characterized by an incremental freeze: every change in the source code requires approval (called a freeze exception). In GNOME, the exceptions are requested by the maintainers (or someone with the corresponding authority) and approved by the release managers and the teams affected by the changes (technical writers, translators or any other project affected by the change). In OpenStack, the exceptions are requested by any developer and approved by the Project Team Leaders.

We observed that there are as many Project Team Leaders or maintainers as projects and teams, and there are as many one-on-one meetings and coordination activities as Project Team Leaders or maintainers. The communication flow is constrained to the organizational structure [23].

5.3.5 (Re)definition of the Product Delivered by the Ecosystem

A common problem that both GNOME and OpenStack ecosystems face is the definition of what comprises the common product; what is is usually called the “official” release. When the ecosystem is driven by the release schedule, and only certain applications are part of the official release, there is an incentive for project developers within the ecosystem to have their applications accepted as part of the official release. The release managers define the official release, and therefore, the criteria to accept new projects (libraries and applications) within the ecosystem to become part of the official release.

We identified that the criteria to accept applications in the official release has been adjusted over time in both ecosystems. For example, one reason to adjust the criteria is to make sure it is clear what the official release is, and avoid diluting the meaning of the ecosystem’s product. Thus, if more than one application provides the same functionality and meets the bare minimum requirements, then the release managers may update the criteria to make clear that only one application per functionality will be part of the official release. For example, having two or three official web browsers or music players, could make GNOME a set of applications rather than an integrated desktop. Similarly, having two or three networking modules providing the same set of features, could make OpenStack a set of libraries rather an integrated cloud platform. However, an unofficial application could replace an official one, but the release managers need to establish a clear criteria to make such replacements.
An official release provides schedule and a list of official applications that comprises, and the official release informs to distributors and integrators of when and what to expect of a release. Hence, applications that are part of the official release are more important than those that are not, and creates a sense of competition between developers of similar applications. The official release creates first and second class projects within the ecosystem and it may alienate some developers who may feel relegated. Such distinction may jeopardize the involvement of companies and developers if the project they contribute to becomes a “second class” project. This distinction is particularly important in OpenStack, where the “OpenStack” trademark can be used only by applications in the official release (“first class” projects).

The challenge is delivering an integrated product rather than a set of applications, and at the same time, defining a set of guidelines and processes for applications to become part of the official release. Both ecosystems have faced these challenges in different ways. In GNOME, the release managers adjusted the rules of how to become part of the official release. It decided that, contributors must propose features instead of proposing applications to become part of the core. These features can be implemented in an existing application, a set of applications, or via developing a new one. As a result, the release managers avoid duplication of features in the official release, and consolidate applications and libraries that offer such features.

The release managers in OpenStack faced an additional challenge. The concept of “official release” made distributors and integrators think that all official applications shared the same level of quality and maturity. In a way, applications that succeed to become part of the official release are expected to be of good quality, or at least of the same level of quality. As Crowston stated [28], success does not imply quality, but the lack of better metrics makes practitioners and researchers use it as an indicator of quality [108].

In OpenStack, the maturity of an application (project) can be associated through the number of deployments and age of a project. The release managers adjusted the rules by dropping the concept of “official release” as a product. The projects still must follow the rules set to become part of the official release, but at the end the release cycle there is no ”official release”. Instead the applications are tagged to let integrators and distributors distinguish the differences between applications. There are three tagging categories: adoption, maturity, and age, which are defined as follows:

**Adoption** is the percentage of production deployments running the project based on the latest biannual user survey results.
**Maturity** is the level of stability and sustainability of the project measured by five parameters, whether or not the project: (1) provides an install guide (2) is supported by seven or more SDKs\(^4\) (3) has an adoption percentage higher than 75% (4) has reached corporate diversity to drive its development, and (5) has stable branches.

**Age** is the number of years since the project started its development.

Some projects have a history of releases and deployment, whereas other do not. The tagging system tries to avoid the perception of first and second class applications, and simultaneously provides information to integrators (and other projects) of the actual status of each application.

Nevertheless, OpenStack is strict in the use of the OpenStack trademark. In the past, only projects in the “official release” could use the trademark. Because of the redefinition of this concept, the release managers adopted another tag for those projects that can use the OpenStack trademark\(^5\). This tag is based in a minimum requirement of adoption, maturity and age.

GNOME had a similar problem in the past. A release manager who was interviewed by Michlmayr [108] stated:

“As GNOME got increasingly bigger it was increasingly like a Linux distribution, ... a big collection of different software that had lots of different maintainers, different technology and maturity. There was always some stuff that was really stable and some that was unstable. You just had to declare a certain day to release and go with it.”

The difficulty of dealing with an increasing collection of applications, with releases combining stable and unstable projects, motivated the developers of the GNOME ecosystem to adopt a time-based release schedule, with specific rules to accept a project as part of the integrated product.

As the ecosystems evolve, the release managers need to redefine the integrated product.

### 5.3.6 Management of Dependencies

In the GNOME and OpenStack ecosystems, release managers have to deal with an ever growing number of dependencies. The number of dependencies grows as the projects that

---

\(^4\)OpenStack considers SDKs “vital” to write cloud applications either for OpenStack or other clouds. The list of known SDKs supporting OpenStack is available at https://wiki.openstack.org/wiki/SDKs.

\(^5\)http://governance.openstack.org/reference/tags/tc-approved-release.html
conform the official release evolve, and when new projects become part of the official release.

There are two types of dependencies that release managers must keep track on: cross-project and external dependencies. A cross-project dependency is a dependency between projects within the ecosystem; for example, libraries. An external dependency is a dependency on projects that are developed outside of the ecosystem boundaries; for example, a library, build tool, compiler, or any tool that developers are required to build or run their projects in the ecosystem. As projects evolve, they are likely to depend on new libraries, or they might develop a tighter integration with other projects within the ecosystem. Whenever there is a new dependency in the official release, the release managers must track its development, increasing the coordination required across projects [18]. Therefore, it is in the release managers best interest to keep the number of dependencies low.

Controlling external dependencies is challenging. The release managers and developers of multiple projects must agree on a specific version of a library required as an external dependency. However, as the external libraries evolve, a project within the ecosystem may have interest in newer versions of such libraries. Then, it is necessary to coordinate which other projects use the library, and to verify the consequences of the changes introduced in the newer version of such library. A positive outcome may happen when other projects have refrained themselves to use new features in order to keep the requirements low (i.e. more accessible for the audience because of a wider availability of older versions). A negative outcome may occur if the new version of a dependency introduces a regression, specifically, when this regression affects other projects with stricter policies regarding regressions.

In GNOME, new dependencies must be discussed to understand the overall impact in the ecosystem better. To accept a new external dependency, the release managers apply the same criteria used to accept projects to become part of the “official release”. For example, an external dependency is in a good stance if (1) it is actively maintained (2) it is widely adopted by distributors (3) it has a reasonable frequency to release new versions, and (4) its developers are quick to respond to issues. The discussion of addition or removal of dependencies takes place in desktop-devel mailing list.

In OpenStack, the release managers provide guidelines for developers before proposing new dependencies for discussion. The guidelines include the bare minimum requisites that developers proposing a dependency must answer. An external dependency: (1) must be actively maintained (2) should be available on distributions that OpenStack targets (3) should offer functionality that no other dependency provides (4) must be released with an OpenStack-approved license (OpenStack prefers Apache2, BSD, and MIT licenses; eventually
they might accept an LGPL licensed dependency) (5) should have good enough quality code (at least, the submitter should inspect the code and make sure the code has tests), and (6) must be compatible with both Python 2 and Python 3. The discussion of addition or removal of dependencies takes place in Gerrit. Developers make changes in a file that tracks OpenStack requirements, and the change is discussed with the release managers, and anyone interested in the outcome. Before submitting a formal proposal, some developers first ask for feedback (request for comments) about their idea in the openstack-dev mailing list; once they have refined their idea, they work on a proposal and submit it in Gerrit.

In both ecosystems, the decision of accepting a new dependency must be evaluated carefully because it has a long term impact. Adding new dependencies is like adding traffic to a highway: it makes the whole system harder to understand and keep under control. Once a dependency is accepted, projects start adopting the features provided by the new dependency later. Therefore, it might be hard to revert the decision and remove the dependency. If the dependency is no longer maintained, the developers must either take over the maintenance themselves, find an alternative project that provides similar features, or remove the dependency (which may add a regression). The control of complexity we found in both ecosystems coincides with Lehman’s second law of software evolution [83]: “As a program is evolved its complexity increases unless work is done to maintain or reduce it”.

In summary, there is agreement in both ecosystems that new dependencies must be careful evaluated before accept them. It may be easier in the short term to address a particular problem by adding a dependency, but in the long term this new dependency can become technical debt, and the cost will be assumed by the ecosystem. On the contrary, removing a dependency helps reduce the complexity of the whole system and may be welcomed by the release managers, and yet it must be evaluated to avoid regressions.

In a FOSS ecosystem, multiple projects and their dependencies add complexity that release managers must keep under control; the practice of discussing and assessing new dependencies make the release managers the stewards of the ecosystem.

### 5.3.7 Continuous Integration and Testing

In both ecosystems there may be large periods with little testing and short periods of extensive testing. To ensure the testing before the time of the release, the release managers set a freeze period to stop the development of new features and start the integration testing. Through the release schedule, the release managers encourage developers to set their

---

6The file requirements within the module openstack. See https://github.com/openstack/requirements.
mindset into bug-fixing.

The FOSS ecosystems we studied require a significant coordination effort because of the size and complexity. The release managers need to align the developers to work on stabilizing their projects for a release. Our findings are in line with what Michlmayr [108, 109] reported for large projects, which may or may not be an ecosystem as described in Section 2.1.2: the coordination in large projects is difficult because participants are volunteers who are geographically disperse. The concept of volunteer used in Michlmayr’s work on release management is interesting because (a) it diverts from the intuitive notion and previous definition of volunteer, and (b) it fits well in the context of our study of FOSS ecosystems. The intuitive notion of volunteer as seen in the literature [38, 43, 80, 106, 111, 112, 114, 117, 127] is a participant who contributes without an economical reward, or in words of Robles et al. [137], volunteers are developers participating “in their free time [and] not profiting economically in a direct way from their effort”. Michlmayr [108, 109] defines the volunteer status based on control, regardless if a developer contributes in the spare time or the time paid by an employer. Participants are treated as volunteers if they are free to work in whatever task they (or they employers) wish. The participants we observed in both ecosystems fit with Michlmayr’s definition of volunteer.

Continuous integration is another major challenge for both ecosystems. Wright [166] reported this challenge as an area of improvement in proprietary software products, and further research is needed.

Continuous integration can affect the ecosystem in multiple ways. When a module cannot be built, it can obstruct the work of every other project that depends on it. It can also obstruct the work of testers, technical writers, and anybody who needs to run the latest development version of the integrated product. It raises the barrier of entry for new contributors who cannot build the development version so they can start contributing. In a FOSS ecosystem there are too many “living” parts that need to be synchronized. Consequently, merging a major feature may break the whole ecosystem, not only when the merge takes place, but also during its integration (secondary effects or regression in other modules indirectly affected).

Modularity enables the development, test, and release of components individually [23]. This can be considered one factor because there is strong link between the structure of the ecosystem and the integrated product developed in the ecosystem. The distribution of tasks between Release Team and Project Team Leaders is an example of such separation. In OpenStack, there are as many Project Team Leaders as projects (modules), who coordinate the release process with the Release Team.
There are issues that can be detected by building separately every single project (module or component). However, a project built correctly still may trigger an issue during the building process of another project that depends on it. This may be introduced by API or ABI changes (either planned or unplanned). This may require high coordination among the projects involved, yet it is an error prone process.

In the OpenStack ecosystem, every project is written in Python. Hence, there is no “build” process. Because of the dynamic nature of the Python language, some errors are triggered during runtime. To reduce the risks of runtime errors, projects rely on coverage unit tests before accepting new contributions. In addition, libraries must be ready early in the release schedule of OpenStack (see Section 4.3), around 6 weeks before everything else.

5.4 Summary

In this chapter, we presented our theory of communication and coordination for release management in FOSS ecosystems. To build our theory, we explored and observed with an open mind the release management process via two case studies on FOSS ecosystems. We identified patterns from these observations, which later we triangulated with the developers themselves, and extracted a set of lessons of release management in practice. We followed a similar approach used by Michlmayr and Fitzgerald [109].

The theory that encapsulates our understanding of the communication and coordination regarding release management in FOSS ecosystems can be summarized as:

1. The size and complexity of the integrated product is constrained by the release managers capacity

2. The release management should reach the whole ecosystem to increase awareness and participation, and

3. The release managers need social and technical skills.

This theory is constrained to the ecosystems in which there is a separation of the technical and administrative roles, that is, there is a Foundation that delegates the technical matters to a group of contributors (technical committee, release managers, or both), there is lack of power over volunteers (the organization does not have a hierarchical authority), and the technical group has autonomy from the administrative one.
The theory proposed in this dissertation can guide future case studies and help the design of controlled experiments, it may also help document the adaptations of this theory to other contexts.
Chapter 6
Conclusions and Future Work

In this dissertation, we presented our study of communication and coordination in the GNOME (cf. Chapter 3) and OpenStack (cf. Chapter 4) ecosystems in the context of release management, and extracted lessons to develop a grounded understanding of release management in FOSS ecosystems (cf. Chapter 5).

We followed a system approach to understand the dynamics of FOSS ecosystems better, and focused ourselves in the challenges of communicating and coordinating in the light of release management.

Any team that develops a software system must plan and manage its releases. In large ecosystems of interrelated independent projects, this task is much more complex than in a single system in GNOME. The Release Team plays a coordination role without participating directly in any particular project. The release management activities are not recorded in commits or pull-requests of any given project, but in discussions in various channels, including email, IRC, and even face-to-face. For researchers studying coordination, their role can be easily overlooked.

In FOSS ecosystems, long time contributors know the inner workings of the projects within the ecosystem, which communication channel to use, who to contact, and the organizational structure. Furthermore, this process might be unknown for outsiders, either practitioners or researchers, who can learn some lessons on how FOSS ecosystems manage releases, how they communicate and coordinate.

This study is limited to release management in FOSS ecosystems, however, our theory may well be applied to some proprietary ecosystems. The literature suggests that the differences between proprietary and FOSS products may be less than generally believed [94, 130]; in particular products with a highly modular architecture [94].

We studied the process of release management in FOSS ecosystems how the process works, and how contributors communicate and coordinate. In the case studies, the release
management is planned, coordinated, and structured in a way that make the releases predictable. Furthermore, contributors will know when a given version will be released even though they might not know which features that version will contain.

Although the release schedule is predictable, it says nothing about the features included nor the success of the product. In other words, the release process can help in “getting the job done”, it may however be the wrong job but it will be done on time. A well defined and planned release process is necessary—although not sufficient—to guarantee the success of the project. Once the release process becomes part of the culture within the ecosystem, it is possible to build upon a solid foundation.

### 6.1 Contributions

The research goals of this dissertation, in the context of FOSS ecosystems, were (1) to understand through an empirical study the release management, (2) to extract lessons and issue recommendations to release managers and those building tools for it, and (3) to build a theory that explains our understanding of release management. To that end, we addressed the following research questions:

1. **Q1.** In the context of release management, how do developers in these FOSS ecosystems communicate and coordinate to build and release a common product based on different projects?
   
   1.1. **Q1.1.** What are the communication channels used for release management?
   
   1.2. **Q1.2.** How do developers communicate and coordinate for release management?
   
   1.3. **Q1.3.** Who are the key actors in the release management process?

2. **Q2.** What are the release management tasks in a FOSS ecosystem?

3. **Q3.** What are the challenges that release managers face in a FOSS ecosystem?

These questions address areas of concern for both release managers as well as software engineering researchers. To gain insight into these areas, our case study interviews focused on these subjects with practicing release managers. The resulting contributions of this dissertation are: (1) an empirical study of release management in two FOSS ecosystems (2) a set of lessons learned from the case studies, and (3) a theory of release management in FOSS ecosystems.

Below is outlined a summary of the lessons we learned from the empirical findings of this dissertation, and a summary of the theory in light of these.
6.1.1 Lessons Learned

We learned common lessons from both GNOME and OpenStack: (1) a well defined release process helps keep the process under control, and (2) the ecosystem needs a common communication channel for coordination. A schedule with well defined milestones facilitates the internalization of the release process by the developers; and once the schedule and release process is known by the developers, the Release Team can focus its efforts on other challenges. Although the Release Team can follow the communication channels used by the developers, it is important for the ecosystem that a common communication channel is used for high level coordination. In both GNOME and OpenStack, the Release Team uses a mailing list to coordinate the release management process within the ecosystem.

From the OpenStack ecosystem we learned that: (1) delegation of release management tasks helps reach the whole ecosystem, and (2) the Release Team must be willing to redefine the official release. The Release Team needs to build awareness of the release process to the whole ecosystem. By sharing tasks and responsibilities with Project Team Leaders, who have the inner understanding of each project, whereas the Release Team has the big picture of the integrated product. As part of the big picture, the Release Team has to assess the concept of “official release”; if it sends the wrong message to the stakeholders, then the Release Team must be prepared to revisit it and re-define it accordingly.

From the GNOME ecosystem, we learned that a Release Team: (1) requires both, good technical and social skills, (2) needs members with a variety of backgrounds, and (3) must follow the main communication channels used by developers. Technical skills are important for understanding the technical aspects of the project, and to earn the respect from their peers; thus, the developers will trust the Release Team’s judgment. Social skills are important to convince developers during the release process, especially because the Release Team lacks official authority or control over them. The diversity of backgrounds in the ecosystem needs some representation in the Release Team to guarantee first-hand knowledge and understanding of the different projects and teams. To reach more developers within the ecosystem, the Release Team needs to communicate using the communication channels used by the developers. In GNOME, the Release Team uses communication channels that vary from asynchronous (such as mailing lists and blogs) to more direct, interactive ones (such as IRC and face-to-face meetings). Asynchronous channels are used to reach a wider audience and enable discussions, whereas synchronous channels are used to answer quick questions, solve coordination issues that involves a few developers, and hold regular meetings.
6.1.2 A Theory of Release Management in FOSS Ecosystems

The result of the empirical studies and set of lessons learned from them is a theory of communication and coordination for release management in FOSS ecosystems, specifically:

1. The size and complexity of the integrated product is constrained by the release managers capacity

2. The release managers should be capable of reaching the whole ecosystem, and

3. The release managers need social and technical skills.

This theory develops a framework for reasoning about the release management process that can be applied by practitioners working on FOSS ecosystems, and for other release managers to understand how globally distributed teams of contributors can coordinate multiple independent projects towards an integrated product.

6.2 Future Work

The work presented in this dissertation can be extended in multiple directions, which we discussed above in the context of one ecosystem. In this section, we present some ideas for further research beyond one ecosystem.

Projects do not exist in a vacuum, neither are the ecosystems we studied. As a future work, it would be interesting to study the relationships between ecosystems.

According to our findings and previous literature [108], GNOME was one of the first FOSS projects to adopt a time-based release schedule, which is also found in other ecosystems, such as Linux distributions (for example, Fedora, Ubuntu and OpenSUSE). It might be interesting to explore to what extent decisions like the time-based release schedule can influence other ecosystems.

We found developers who act as bridges between different projects across the ecosystem—they are facilitators that enable projects to move forward together and towards a common goal. As projects in an ecosystem are not developed in a vacuum, we suspect that software ecosystems interact with other ecosystems. If this is the case, we would also expect to find 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. For example, FreeDesktop is an initiative to encourage sharing common libraries in projects such as GNOME and KDE\textsuperscript{1}. It might be interesting to study those projects that are common between ecosystems and the way they communicate and coordinate to pursue their own goals. A project in a similar situation is the FOSS Web rendering engine WebKit\textsuperscript{2} that is used by different vendors in FOSS and proprietary contexts.

Analogously, GNOME releases are made available to the users through software integrators known as distributions. Those software integrators are also projects and might be ecosystems themselves, such as the Fedora Project or OpenSUSE. They also have their own Release Teams that could prove to be interesting subjects of future research. Both Fedora\textsuperscript{3} and OpenSUSE\textsuperscript{4} are Linux distributions.

In other words, projects that produce software are known as *upstream* and projects that consume their software *downstream* [1].

In OpenStack, we found team leaders who also contributed as packagers in Linux distributions, for example, in Debian. Those developers may be the bridges between ecosystems.

A question that arose during the development of this dissertation was: is the ecosystem shrinking or expanding? How have the communications in the ecosystem changed over time?

It could be beneficial for projects to address any potential problem. Ecosystems can be perceived as large-projects, therefore, a long term tendency could not be perceived as in small projects where small changes in the dynamic of their developers can be noticeable.

\[\text{\footnotesize\textsuperscript{1} Similar to GNOME, KDE also provides a FOSS GUI desktop system} \]
\[\text{\footnotesize\textsuperscript{2}http://webkit.org/} \]
\[\text{\footnotesize\textsuperscript{3}http://fedora.org/} \]
\[\text{\footnotesize\textsuperscript{4}http://opensuse.org/} \]
Bibliography


ference on Weblogs and Social Media (ICWSM 2009), pages 361–362, May 2009. 3.1.4


[16] A. Bohn, I. Feinerer, K. Hornik, and P. Mair. Content-Based Social Network Analysis of Mailing Lists. The R Journal, 3(June):11–18, 2011. 2.3.2


[27] K. Crowston and J. Howison. The social structure of Free and Open Source software development. *First Monday*, 10(2), 2005. 4.5, 4.5.1


[75] E. Knauss, D. Damian, G. Poo-Caamaño, and J. Cleland-Huang. Detecting and classifying patterns of requirements clarifications. In *Proceedings of the 20th Inter-
national Requirements Engineering Conference (RE 2012), pages 251–260. IEEE, Sept. 2012. 2.3


[108] M. Michlmayr. *Quality Improvement in Volunteer Free and Open Source Software Projects Exploring the Impact of Release Management*. PhD thesis, University of Cambridge, Mar. 2007. 1, 1.1, 1.2, 2.2, 2.2.1, 4.4.3.1, 5.2.2.2, 5.3.4, 5.3.5, 5.3.5, 5.3.7, 5.3.7, 6.2


Asia-Pacific Symposium on Visualization (APVIS 2007), pages 25–32. IEEE, Feb. 2007. 2.3.4


[129] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972. 5.2.1, 5.3.1


[154] The GNOME Foundation. GNOME Foundation Charter Draft 0.61. http://foundation.gnome.org/charter.html, 2002. 2.4, 3.2.1, 3.2.1


Appendix A
Human Research Ethics Board Approval

This research was submitted to the University of Victoria Human Research Ethics Board under the titles “Development Process in Free/Open Source Software Projects” and “Development Process in Free/Open Source Software Projects: A Qualitative Study”, which were given protocol numbers 12-023 and 12-024, respectively. The principal investigator was Germán Poo-Caamaño, and the supervisor was Daniel M. German. The start date of the research was January 20, 2011; and the expiration date approved was January 18, 2017. The research was approved by Dr. Rachael Scarth.
Appendix B
Interviews

This appendix contains the interview questions we performed both in person and via email.

B.1 In Person Interviews

Below a semi-structured interview questions for in-person interviews that we conducted during a conference.

There are 3 categories of questions in total. Please, provide as much or as as little detail as you like. “yes/no” is fine, but if you have an interesting insight or story to share, I would greatly appreciate it. If you feel I have missed an important aspect that a question is leading, please feel free to add additional information outside the questions

Consent

Please choose one of the following two options regarding to ANONYMITY of your data:

1. Full ANONYMITY: We will not use your name during the analysis or when reporting results.

2. Waive your right to ANONYMITY: In this case, we will acknowledge you in a any quotation we use.

B.1.1 General questions

B.1.1.1 Use of communication channels

In your role as a software developer:
1. What kind of communication needs do you have to collaborate with other projects in GNOME?

**B.1.1.1.1 Information consumer.** When your project depends on other projects:

1. Do you need to learn or know from other projects?
   
   (a) How many projects do you need to be aware of?
   
   (b) How do you keep yourself aware of the activities in the other projects? Can you give some examples of how that happens?
   
   (c) Are the communication channels good enough for communicating with other projects?
      
      i. How do you think the communication can be improved?

**B.1.1.1.2 Information producer.** When other projects depend on yours:

1. Do you need to learn or know from other projects?
   
   (a) How many projects need to be aware of the activities in your project?
   
   (b) How do you make others aware of the activities of your project? Can you give some examples of how that happens?
   
   (c) Are the communication channels good enough for communicating with other projects?
      
      i. How do you think the communication can be improved?

2. When you need to communicate with other project developers, how and when do you use the following communication channels:

<table>
<thead>
<tr>
<th></th>
<th>When</th>
<th>How</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bugzilla</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct email</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mailing list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phone/Video conference</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
B.1.1.2 Roles

In the software development process there are different roles inside a project (e.g. maintainer, sporadic contributor, translator, interaction designer, among others):

1. What are your roles in the projects you work on?
2. Does the role affect the way you communicate with other projects?
3. Are there pre-defined guidelines to communicate others? For example, announcements seems to follow a common pattern in the titles, content structure and communication channels used to make the public. Could you give more examples likes that?

B.1.1.3 Conflict resolution

In your communication with other projects

1. How do you coordinate activities?
2. How do you organize the schedule and the planning across projects?
3. Do you face conflicts?
4. How do you resolve them?

B.1.2 Study-related questions

B.1.2.1 Activities, roles and communication channels

In the following table, the columns contain activities that can be associated to a project (roles or teams you participate).

- Please, write a rough percentage of the time you spend on each activity that fulfill what you do.
The following table contain the same activities, this time the rows contain the communication channels that can be used for each activity.

- Please, evaluate the importance of each communication channel has for every activity you perform. Use the following values: (1) Intensive use (primary or critical), (2) regular use (secondary or important), (3) light use, and (4) infrequent use.

<table>
<thead>
<tr>
<th>Development</th>
<th>Documentation</th>
<th>Translation</th>
<th>Usability</th>
<th>Art-work</th>
<th>Accessibility</th>
<th>Release</th>
<th>QA</th>
<th>Marketing</th>
<th>User groups</th>
<th>Sysadmin</th>
<th>Board</th>
<th>Advisory board</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mailing lists</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IRC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bug Tracker</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit logs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wiki</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Blogs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hackfests</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conferences</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared docs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In our study we have discovered 9 types of communications in the mailing list *desktop-devel-list*.

- Do you think there is another one that is missing?

- With respect to ones we discovered: In the left side of the table, please indicate the percentage of participation you think you participate in each one. In the right side of the panel, please indicate the communication channels do you use for each type of communication (you can mark as much as you like).
## B.1.2.2 Interaction with other project’s developers

- Why do you interact with developers of `<project/team_name>`, which seems different to the project you work on?
  
  - *Example:* Why do you interact with people from the accessibility, which seems different to the project you work on? (You do not work on accessibility)
  - Are you acting as an intermediary of the `<project/team_name>`?

- Have you acted as an intermediary of other projects or developers? *(that are not in the graph)*
  
  - In which circumstances? Why?

- It seems you are part of `<project/team_name>`, where some members are represented in the graph talking to different group of people each.
  
  - Why do you talk with a particular group of developers and not with the others?
  - Is this something you have agreed in advance in `<project/team_name>`?
• Given your role as `<role_name>` in `<project/team_name>`, I would expect you to communicate with `<projects/developers_name>`, but your communication seems localized in a small cluster or it seems you are missing some developers.

  – Are your communication needs different than the ones I was expecting from what I studied?
  – Do you use a different communication channel to approach the missing ones?

• According to the graph, you communicate with a small set of developers (*peripheral communicator*):

  – Are your communication needs focused on that set of developers?
  – If not, is there anything that could be improved in the communication channels or anything else?

### B.2 Via email Interviews

We used the experience obtained running the in person interviews to improve the understandability of the questions. After a first unsuccessful email interview, we shortened the length of the interview and focused the questions that had provided more information from the in person interviews. We also cut down the question that were hard to perform via email, for example, the ones that require to interact back and forth with the interviewee.

We sent the interviews together with the introductory recruiting text. To increase the odds of obtaining a response, we elaborated the introductory text to show that we performed a previous research. We also provided a short version in the jargon used by some FOSS developers (*TLDR*¹) of the Implied Consent.

Below the introductory text and the interview questions for email interviews that we conducted.

**Introductory Recruiting Text**

Hi {Participant},

I have been studying the release management process in FOSS projects for the past 4 years. Most of my analysis has involved analyzing mailing list archives,

¹*TLDR* stands for “Too Long, Did’n Read”, which is like a summary to show the main points of a long text.
documentation in the wiki, etherpad, and blog posts. In order to triangulate my findings, I have been interviewing FOSS developers.

Since you have been involved either in the Release Team, Project Team Leader, or both. I would greatly appreciate if you could answer inline the questions, or if you prefer, answer them via other media (e.g. Skype, Hangout, telephone) for 10 to 20 minutes.

OpenStack is specially interesting to me because it is an ecosystem of projects. Projects that require some coordination towards a common release. I would like to understand how the many projects communicate and take decisions.

Thanks, Germán.

Interview Text

There are 3 categories of questions in total. Please, provide as much or as as little detail as you like. “yes/no/it does not apply” is fine, but if you have an interesting insight or story to share, I would greatly appreciate it. If you feel I have missed an important aspect that a question is leading, please feel free to add additional information outside the questions

Implied Consent

By filling in this email interview and sending it to me, YOUR FREE AND INFORMED CONSENT IS IMPLIED. Please see the bottom of this email for more information regarding this study. Please choose one of the following two options regarding the ANONYMITY of your data:

1. Full ANONYMITY: We will not use your name during the analysis or when reporting results.

2. Waive your right to ANONYMITY: In this case, we will acknowledge you in any quotation we use.

B.2.1 Use of communication channels

Please keep in mind that you might be an information consumer (your project depends on information from other projects), an information producer (other projects might need information about your project), or both.
1. What kind of communication needs do you have to collaborate with other projects in OpenStack?

2. How many projects do you need to be aware of? / How many projects need to be aware of the activities in your project?

3. How do you keep yourself aware of the activities in the other projects? / How do you make others aware of the activities of your project?

4. When you need to communicate with other project developers, how and when do you use the following communication channels: IRC, Instant messages, Launchpad, Mailing list, Wiki, Etherpad, Direct email, Phone/Video conference, any other?

5. Are the communication channels good enough for communicating with other projects? How do you think the communication could be improved?

B.2.2 Roles

In the software development process there are different roles inside a project (e.g. maintainer, project leader, sporadic contributor, interaction designer, among others):

1. What are your roles in the projects you work on?

2. Does the role affect the way you communicate with other projects?

3. Are there pre-defined guidelines to communicate others? (e.g. announcements seem to follow a common pattern in the titles, content structure and communication channels used. Could you give more examples likes that, either official or not?)

B.2.3 Conflict resolution

In your communication with other projects

1. How do you coordinate activities?

2. How do you organize the schedule and the planning across projects?

3. If you face conflicts, how do you resolve them?
**Implied Consent (detailed)**

*TLDR:* Your participation is important for this study, but it is voluntary. There is no risk identified by participating in this study. Nevertheless, we will protect your confidentiality. The information recorded will be disposed within 3 years. The results will be disseminated in multiple venues. If you have any concern, please contact the people indicated below either by email or phone.

You are invited to participate in a study entitled Development Process in Free/Open Source Software (FOSS) Projects that is being conducted by Germán Poo-Caamaño and Daniel German.

Germán Poo-Caamaño is a graduate student in the department of Computer Science at the University of Victoria and you may contact him if you have further questions through email gpoo@uvic.ca or by phone +1-250-472-5877. As a graduate student, I am required to conduct research as part of the requirements for a PhD in Computer Science. It is being conducted under the supervision of Dr. German. You may contact Dr. German through email dmg@uvic.ca or by phone 250-472-5790.

*Purpose and Objectives.* The purpose of this research project is to systematically and comparatively examine development practices used by successful Free/Open Source Software projects and to understand how these practices compare to those used in other settings, such as industrial development. Using automated analysis, we can extract the basic quantitative parameters regarding contributions. The purpose of the current study is to collect data pertaining to parameters that cannot be extracted automatically from a mailing list, bug trackers, software repositories, and websites; to discuss how the group interacts while contributing to a project, and to ensure that our results make sense to FOSS practitioners. We are also interested in the development policies used by the project and how these were developed. The contribution of this work is to deepen our understanding of communication and collaboration in a Free/Open Source Software development and suggest how successful practices might be incorporated into industrial development. We are not seeking to collect identifiable information about third parties.

*Importance of this Research.* Unlike other types of engineering, software engineering is still developing processes and metrics that yield timely, high-quality products. Overly formal, management driven processes are not always the solution. We want to understand how development, communication and decisions are conducted in the FOSS community.

*Participants Selection.* You are being asked to participate in this study because you are an FOSS developer who has in the past had or who currently has a regular contribution in
an FOSS project.

What is involved. If you agree to voluntarily participate in this research, your participation will include answering this email interview questions (10-20 minutes) and optionally participating in a telephone interview. The audio from the telephone interview will be recorded and stored in a password protected file.

Risks. There are no known or anticipated risks to you by participating in this research.

Benefits. The potential benefits of your participation in this research include reflection on the development process that your project uses, a better understanding of the development practices used by FOSS projects, and the potential for software projects to interact with other actors (industry, other disciplines interested in FOSS, etc.)

Compensation. There is no compensation.

Voluntary Participation. Your participation in this research must be completely voluntary. If you do decide to participate, you may withdraw at any time without any consequences or any explanation. If you do withdraw from the study, your data will not be used in the study and will be deleted.

Anonymity. There are two possibilities regarding anonymity. Please see the beginning of this email to select the level you desire.

Confidentiality. Your confidentiality and the confidentiality of the data will be protected by keeping it in a password protected database that only the research team has access to.

Dissemination of Results. It is anticipated that the results of this study will be shared with others in the following ways: published in articles, presented at scholarly meetings, and published in a thesis. Also, the results of this study will be given back to each project with anonymized data.

Disposal of Data. Data from this study will be permanently deleted within three years of collection.

Contacts. Individuals that may be contacted regarding this study include Germán Poo-Caamaño (gpoo@uvic.ca) and Daniel German (dmg@uvic.ca).

In addition, you may verify the ethical approval of this study, or raise any concerns you might have, by contacting the Human Research Ethics Office at the University of Victoria (250-472-4545 or ethics@uvic.ca).

By filling in this email and sending it to me, YOUR FREE AND INFORMED CONSENT IS IMPLIED and indicates that you understand the above conditions of participation in this study and that you have had the opportunity to have your questions answered by the researchers.
Appendix C

Application for Qualitative Analysis

This appendix contains a description of the application we built to assist us during the qualitative analysis of the mailing list discussions.

Figure C.1 depicts the user interface of the application.

The left pane contains the list of correlative threads, and whose purpose is to facilitate the navigation between discussions. The pane also indicates whether a discussion cross–project or specific, and the categories already assigned. A discussion is considered cross–project is at least one of the messages in the discussion contains a topic that is cross–project. That is, in mailing list discussions developers may change the topics of the messages if they think the discussion should continue with a wider or narrower audience.

The middle pane is divided in two areas mimicking an email reader: (1) a nested list of messages in the thread, the date when the message was sent, and a message indicator for cross–project messages; and (2) the content of each message. The messages are threaded using Jamie Zawinski’s algorithm ¹

The right pane contains the list of categories. The researcher can select a category from the list or add a new one using the entry box on the top.

The bottom pane has additional information, such the dates of the first and last email in the discussion, the duration of the discussion in days, the name and email of the leader of the discussion (the person who started it), the number of messages, the number of participants, and the topics detected in the discussion. Finally, an space for entering additional comments during the coding process.

---

¹https://www.jwz.org/doc/threading.html
Figure C.1: Application used for coding and abstracting themes from the mailing list discussions.
Appendix D
Contributions

This appendix contains a list of contributions in peer reviewed publications related to this dissertation.


