UNIVERSITÄT PASSAU

Fakultät für Informatik und Mathematik

Bachelor's Thesis in Computer Science

Communication of Core and Peripheral Developers in OSS Projects: An Exploratory Study

Sofie Kemper

2017-09-18

Advisors:
Prof. Dr. Sven Apel
Thomas Bock
Claus Hunsen
(Chair for Software Engineering I)

Kemper, Sofie:
Communication of Core and Peripheral Developers in OSS Projects: An Exploratory Study
Bachelor’s Thesis, University of Passau, 2017.

Abstract

As the developer community in open-source software (OSS) projects typically experiences little stability and may be geographically dispersed, coordination and communication among the developers are crucial for the success of the software project and, thus, warrant further investigation. In this thesis, we perform a long-term, empirical study of the communication in eleven open-source software projects. We investigate how the notion of core and periphery, i.e., the developer's activity level regarding source-code contributions, is reflected in the development-mailing-list communication. We investigate not only the activity of the participant groups but focus mainly on their interaction as this has not yet been extensively studied.

We find that core developers are generally the most active mailing-list contributors, but that non-developers amount for a surprisingly large proportion of the project communication. In addition, we observe a very strong preference towards core members as communication partners in all contributor groups. The strength of this preference as well as the classes' activity level is project-specific but stays very consistent over time within projects. Moreover, we observe that pair-wise core-core communication exhibits great longevity but is generally less intensive than core-peripheral communication. Peripheral-peripheral communication is characteristically short-lived. Our data does not indicate that project communication changes qualitatively during the project release cycle. In general, we observe that the project core and periphery differ not only in their activity levels but also in how they interact on the project's mailing list, i.e., their preferences for communication partners as well as the longevity and intensity of their interactions.

Contents

List of Figures

List of Tables

1 Introduction

Human factors – most notably the collaboration between developers – are an important factor in whether a software project succeeds as well as determining the quality of the resulting software [17]. Large-scale, complex software projects, in particular, require a great amount of communication and coordination amongst the developers to guarantee that a coherent product is produced [1, 17]. Although it might be possible to impose certain coordination requirements, e.g., pair-programming or weekly meetings, and, thus, guarantee adequate developer communication in traditional commercial/closed source software (CSS) development [13], it is much more difficult when regarding open-source software (OSS) projects [17].

OSS development has become an important part of the IT ecosystem and is typically characterised by a very large, ever-changing developer community [13]. Additionally, one notable characteristic of the social community of an OSS project is its self-organising nature – its structure is not imposed by any project leader but continually adapts to the development coordination needs of the corresponding project. Hence, it is difficult to ascertain that communication needs are met [17]. As this developer coordination is an important aspect in the success of the software project, it warrants a closer examination.

The communication between *core developers* – those who contribute continually and extensively to the software project in question – and other developers is particularly interesting, as its characteristics remain mostly unstudied. Since new members are extremely important for OSS projects due to the high developer turnover, it is interesting in which capacity and how they communicate with the more experienced contributors. This can serve as an indicator of how well new developers are integrated in the project [6, 13]. Moreover, the health of its developer community directly influences the success of an OSS software project as potential users may choose software not only based on its features but also on the provided support which emerges from the virtual community surrounding the project [25].

Understanding OSS developer communication, in particular the communication between core developers and developers new to the project, might help us to recognise and combat deficiencies, e.g., coordination requirements that are not met [17]. Similarly, it might be possible to formulate general guidelines for the communication in OSS development as well as finding indicators of developer communities' health [13, 6]. As the global E-Commerce structure heavily relies on OSS, e.g., Apache web servers as well as the Linux and BSD operating systems, a better understanding of how OSS developer communities function might help IT planners in establishing trust and developing more effective strategies regarding the usage of the corresponding software [23]. Thus, we hope to gain a deeper understanding of OSS developer communities through this study.

In this thesis, we perform a long-term empirical study of the communication in eleven open-source software projects, focusing on how the notion of core and periphery, i.e. the developer's activity level in the source code, is reflected in their communication on the mailing list. We analyse not only the activity of the participant groups but additionally investigate the nature of their interactions, thus, supporting the evidence found in prior research that the developer groups differ not only in their activity levels but also exhibit qualitative differences in their behaviour.

We find that core members are generally the most active mailing-list contributors. However, non-developers amount for a surprisingly large proportion of the project communication. A very strong preference of core members as communication partners is visible in all contributor groups in all eleven subject projects. Although the distinctness of this preference as well as the classes' level of involvement in the mailing list varies greatly among our casestudies, it stays very consistent over time within projects. Moreover, we observe that pair-wise core-core communication exhibits great longevity but is generally less intensive than core-peripheral communication. Peripheral-peripheral communication is characteristically short-lived. Our data does not indicate that project communication changes qualitatively during the project's release cycle.

This thesis is structured as follows: We start by presenting necessary background information in Section 2. This includes related work that has been published in this area of research as well as the theoretical basis for our study. For this purpose, we introduce some characteristics of OSS projects, social networks representing developer collaboration, and the classification of developers into *core* and *periphery*. In Section 3, we explain our study design starting with our research questions. Then, we present the independent, the dependent, and the confounding variables of our study. Thereafter, we introduce the hypotheses as well as some characteristics of the chosen casestudies and our implementation. Consequently, we present our findings concerning each hypothesis in Section 4. In Section 5, we discuss these results and disclose threats to the validity of our findings. We conclude our work with a summary of our study and results as well as suggestions for future work in Section 6

2 Background

In this section, we present the theoretical basis for our study. We begin by introducing some characteristics of OSS development and its developer communities. We further explain the social networks representing developer collaboration as well as giving a brief overview of how they are constructed. Then, the theoretical model of developer roles as well as the practical classification of developers into *core* and *periphery* according to count-based and network-based metrics are introduced. Additionally, we present some related work and situate our study in relation.

2.1 Characteristics of OSS Projects

Open-Source Software (OSS) development has become an important part of the IT ecosystem and, thus, merits a closer examination. OSS projects are well-suited for studies of software development processes and developer coordination as their version control systems as well as their mailing lists are archives of the project history, providing all past correspondence as well as the source-code authorship history [2]. Although OSS projects can vary greatly – particularly in size, organisational structure, and the composition of their developer community – there are some common characteristics that apply to almost all OSS projects to some extent, manifesting in different ways [31]. When investigating OSS developer communities, it is important to keep these properties in mind to correctly interpret the findings.

Open Source Software or *Free Software* designates software products which are made available to their users under a license which allows them to be freely studied, modified, derived from, and redistributed. Thus, this software development model is a major deviation from traditional or *Closed Source Software* (CSS), as OSS projects allow anyone (read-)access to the software's sources [31]. This is based on the assumption that, by sharing source code, the contributors cooperate within a model of rigorous peer-review and that the parallel development process is ideal for evolving software products [18]. Although the OSS development model has been used for decades, its importance grew with the internet's rise in popularity and, especially, the emergence of E-commerce [23].

An OSS project is unlikely to flourish unless it is accompanied by a healthy developer community which provides the basis for developer and user collaboration [33]. One of the characteristics of OSS projects is their lack of control over the project contributors [31]. An OSS project's developer community typically consists of volunteers which are led to contribute by a combination of intrinsic and extrinsic motives [30]. These developers are not assigned to teams by a common manager, but volunteer instead to participate and contribute to certain tasks [29]. This can lead to a continually shifting membership and high turnover in the developer community [10]. The developer community is characteristically geographically distributed and larger than that of a comparable CSS project as its members usually do not spend all their time working on the project [31, 18].

In contrast to CSS projects, the OSS project's developers are typically also users of the software and all users are potential contributors. The symbiosis of the developer and user community provides a quality control by use, which has been shown to perform favorably for evolving software projects [10]. In many cases, there are no explicit requirement documents as the developers – through usage of the software – provide the requirements and can self-assign these tasks [31]. In general, the participants employ an autonomous mode of decision-making through which they define and work on their own tasks as no central management with control over the developers' activities exists [10, 31]. Studies have indicated that OSS projects benefit from this task-assignment process as it makes a developer's productivity independent of the number of project developers – a violation of *Brooks' law* established for CSS projects [18, 17].

Although different OSS projects possess different organisational structures, e.g., the "benevolent dictatorship" of LINUX which is in stark contrast to the well-organised hierarchical governance structure of LUCENE [11], its developer communities typically self-organise. One commonly employed imagery is the comparison of OSS structures to an ever-changing "bazaar" without pre-designed or explicit structure while equating CSS projects to organisational "cathedrals" [2]. Placing communication at the center of this collaboration mechanism, the structure typically adapts dynamically according to coordination requirements, the source code structure, and development tasks [7, 17]. Thus, the project may benefit from this lack of fixed organisational structure as it allows for an ideal alignment of the software development and the developer community [2].

The current version of the source code is at the core of any OSS project and its knowledge base. Nonetheless, this knowledge base is also mirrored and complemented in other parts of the project's organisation system such as the mailing list archive, the documentation and bug reports [10]. Additionally, this partitioning is reflected in the project tasks: Despite the production of code as apparent core task, other duties such as providing documentation, testing and evaluating existing parts of the system as well as providing user support all form valuable and necessary contributions to the project [2, 10].

Due to the lack of traditional management structure, communication is a crucial factor in the developer coordination and, thus, the success of the software project [1]. Characteristically, conversations in OSS settings are conducted in a public manner and archived for later reference. For this purpose, *mailing lists* (MLs) – of which there may exist several for a multitude of different project matter discussions, e.g., development, patches, user support – are used [11]. Mailing lists are public forums to which anyone can post messages and, thus, allow people of all experience levels to communicate about practically any topics pertaining to the OSS project [1, 13]. Although several projects supplement these mailing lists with other communication channels, such as issue trackers or features of the development platform, e.g., discussions of pull requests on GitHub, mailing lists are traditionally considered the hub of project communication [8, 11]. By community norms, these lists host all discussions regarding the system and development tasks [2]. Admittedly, many projects shift the discussions and developer coordination to other channels of communication in recent years. Nonetheless, newcomer's first interaction with the project's community will usually take place on the mailing list and they constitute a public archive of past decisions and the communities culture [13]. Thus, we expect them to give valuable insights into an OSS project's community structure.

The source code – as indicated by the name *open*-source development – is always available on the Internet, mostly in a publicly-accessible *version control system* (VCS). Although the write-access to the code may be limited to a privileged group of contributors, every potential contributor has read-access to the repository and can provide his contributions in the form of patches to the mailing list or by using features of the VCS hosting platform, e.g., forks and pull requests in GitHub [31].

2.2 Developer Networks

One of the most natural and accurate ways of representing human interaction in the form of developer communities are developer networks. This type of social network as well as the network construction mechanism employed in this work are presented in the following.

2.2.1 Developer Networks for OSS Community Analysis

Many parts of nature and culture – human interaction, in particular – are organised in networks and, hence, can be represented and studied very accurately via these structures [4]. Especially, social network theory has received considerable attention in the research community. Social scientists theorise that individuals are embedded in webs of social interactions and relations which can be represented ideally through network structures. One of the most important aspects that are examined with regards to these social networks is the question how persons combine in networks, i.e., how they interact and relate to one another, and how these interactions provide the basis for enduring, functioning communitites [5]. Additionally, one of the most potent ideas in social network analysis is the notion that the outcome of individual actors, groups of actors, and the network in its entirety can be explained – and even predicted – through measurable network properties and the actors' relative positions in the structure [4].

Due to this natural representation of community structures via social networks, developer networks have found many applications in studies of developer communities, e.g. in the work of Bird et al. [1] and the studies of Joblin et al. [17]. Developer networks are *socio-technical* networks, meaning that the developers' mutual activities are technical in nature but the network constructed from these activities represents the corresponding relationships between the developers [17]. These networks have been shown to accurately reflect developer perception of developer communities' structure as well as its properties [16]. For many aspects of interest, a network perspective can yield a richer understanding than mere count-based metrics, e.g., the number of mails written between two individuals. [16].

The formal representation of these developer networks is a graph whose *nodes* (or *vertices*) constitute the actors – i.e., the individual developers – and whose *edges* (or *ties*) form the relations between these actors, typically representing some type of collaboration among them. This collaboration can either be direct, e.g., mails sent from one developer to another or one replying to issue tracker comments of the other, or indirect, e.g., two developers working simultaneously on the same source-code artifact or posting to the same threaded mailing list discussion [17]. Nodes as well as the relations among them may possess attributes [4]. In the investigation of developer communities, exemplary vertex attributes could be the developer's age as well as his seniority or status in the project. Similarly, edges in developer networks could be characterised via their creation date (i.e., the date of the corresponding developer interaction) or their importance.

Using developers networks, we can investigate different aspects of the developer community. In general, three levels of analysis are distinguished in social network theory: (1) the study of individual edges – the *dyad level* – constitutes the most fundamental level of analysis [4]. This would allow us for instance to investigate whether developers that have participated in the project for similar times are more likely to collaborate. (2) The second level – the *node level* – consists mostly of aggregations of dyad-level properties such as the actor's number of ties [4]. A sample research question addressable at the node level could be: "Is the number of actor's outgoing ties positively correlated to the probability of them staying active in the software project?" (3) The third and most general level – the *network level* is about analysing the network as a whole [4]. One application might be asking how densely or sparsely connected the developer network is and in what ways this impacts the community's stability.

2.2.2 Construction of Developer Networks

To understand the implications of our findings, it is important to explain the mechanisms we use for constructing our developer networks. For this work, we consider two types of developer networks – *email social networks* and *source-code collaboration networks*. Although we mainly analyse the social networks that are constructed based on the OSS project's mailing list, we enrich our understanding of the mailing list data by using networks that represent the developer collaboration on source code artifacts. Both classes of networks are constructed in the same way, but based on different artifacts – either source code artifacts or mails organised in threads.

For the construction of *email social networks*, we consider only the project's primary development mailing list. We assume that two developers communicate, i.e., should be linked via an edge in the network, when they post to the same mailing-list thread. To accurately represent the information flow as well as the "direction" of messages, i.e., which developer sends a message to which other developer(s), we use a construction mechanism which, for each mail written to a certain thread, establishes connections going from the mail author to the authors of all previous contributions in this thread. As an example, if A initiates a thread, B replies and C writes a third message, the resulting network will contain edges $B \to A$, $C \to A$, and $C \to B$. We show an exemplary network which contains these edges in Figure 2.1

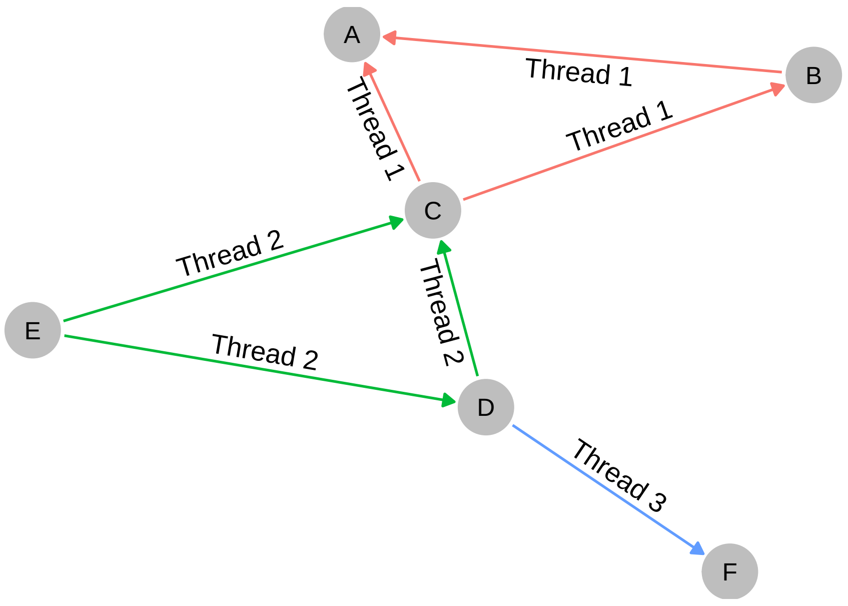

Fig. 2.1 Examplary Mail-Based Developer Network

Source-code networks are constructed analogously, but using the history of changes made to a source-code artifact instead of email threads. We only consider the commit authors in the main development branch and construct the networks based on *function-level artifacts* as these have been shown to produce networks that agree with developer perception [16]. Hence, if developer D_2 makes changes to a function to which developer D_1 has previously contributed, the resulting cochange network will contain an edge $D_2 \to D_1$.

We use this construction technique, as it allows us to represent not only the *existence* of interaction among developers, but also the direction of the information flow in this interaction. Through this, we gain a more accurate representation of the chronological sequence of collaboration events [1, 6]. For details of the data extraction process as well as the interpretation of the constructed networks, the interested reader is referred to Joblin et al. [16, 17].

2.3 Core/Periphery Classification

The notion of *core* and *periphery* is an important one in social network theory and seems to be a prevalent structure in human social networks [5]. OSS developer networks, in particular, have been shown to follow this model, allowing researchers to divide their members into a large, volatile peripheral group and a significantly smaller group of core members [17]. The notion of core and periphery in OSS was born from a model of OSS participation roles, both of which will be explained in the following. Additionally, we explain which operationalisations of the developer roles are used in this work.

2.3.1 Model of OSS Project Participant Roles

Although OSS developer communities may not possess any strict hierarchical organisation, their structure ist not completely flat. The project members have different levels of influence on the system based on the roles they play [33]. With regards to developer roles, the main difference between OSS and CSS development is the role transformation of the stakeholders of the project. In CSS projects, roles are typically assigned by managers and rarely change; Users and producers are two clearly separated groups [33, 13]. In OSS, due to the partial overlap and close contact between the developers and the users, roles change constantly. Users may start to contribute and members may change their roles based on how much they want to get involved in the developer community [10, 33].

There are different approaches of distinguishing among different developer roles. One easy classification metric is whether the developers have write access to the project's VCS. However, this formal list of developers usually does not present an accurate representation of the contribution of project members [10]. Thus, more complicated structures were introduced. One of the most common models of these roles and role transitions is the *onion model*. Although other models or refinements, such as the onion pyramid suggested by Jensen and Scacchi [14], have been proposed, the onion model is still widely used.

The onion model is based on eight roles that the stakeholders in the project can assume. These roles – as illustrated in Figure 2.2 – range from passive users to core developers and indicate concentric levels of distribution where roles closer to the center possess a larger radius of influence [31, 33]. Thus, the model corresponds to layered organisational structures prevalent in management, economics, and software engineering literature [14].

Research has shown that, although not all eight types of roles exist in all OSS communities and the exact community composition may vary, the

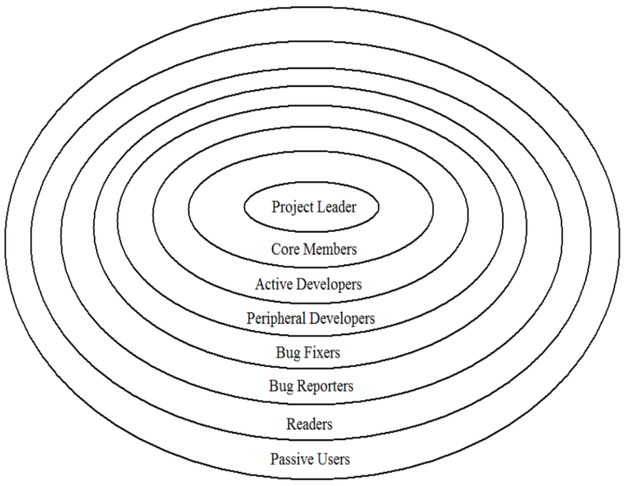

Fig. 2.2 The Onion Model, taken from Jensen et al. [13]

onion model aptly reflects the structures in OSS projects and substantial size differences among the eight roles exist as indicated in Figure 2.2 [33, 16]. OSS developers have been observed to gravitate towards more central roles with prolonged involvement in the project [14]. For the success of OSS projects, it is important that the path from passive user to core member is possible, although not many members of the project will follow this idealised path [33].

This advancement of roles is typically meritocratic in OSS developer communities – members advance by proving themselves through continued, valuable contributions and social commitment to the project via involvement on the mailing list, thus, gaining higher recognition in the developer community as well as more responsibilities and more central roles [14, 33]. The formality and explicitness of this advancement process – especially the access to the VCS – depends on the structure and size of the software project [31]. An interesting observation regarding this advancement process is that, as members change roles, they change the social dynamics of the project and, thus, reshape the structure of the community [33]. Therefore, the role advancement processes are important not only for the individual members and their motivation, but also for the evolution of the community as a whole.

2.3.2 Core and Periphery in the Developer Community

The onion model with its eight different roles has been simplified to the dichotomy of *core* and *periphery* [16]. In most OSS projects, one can observe the harmonious coexistence of a stable group of core members and a fluctuating periphery [17].

The distribution of activities in OSS projects is generally highly skewed with a very small fraction of the members responsible for a majority of contributions [10]. This participation inequality has been observed in the contribution of source code as well as the distribution of traffic on the mailing lists [8]. The phenomenon is very common in OSS projects and increases with the system size [17].

Thus, the core/periphery dichotomy is based on the participation inequality in OSS projects. Additionally, it reflects the network theory structure of combining a dense, cohesive core with a sparse, unconnected periphery [3]. In some projects, such as LINUX, the core/periphery dichotomy will even be reflected in a tiered structure of mailing lists [20]. The two roles are defined via certain characteristics and can be operationalised in different ways as we describe in Section 2.3.3.

Core developers are characterised via their prolonged, consistent and intensive involvement in the project. As they typically have an extensive knowledge of the system architecure, they play an essential role in developing the software. Additionally, they can exert a strong influence on project decisions and, thus, form the general leadership structure of the project [16, 17]. Research suggests that the core consists of only a very small fraction of project members, combining the two classes *project leader* and *core member* in the original onion model (see Figure2.2) [8, 13].

Developers that are part of the *periphery* are distinguished by a rather short-term, irregular involvement. They typically provide smaller contributions, such as bug fixes or smaller enhancements of the system [16]. Most members of an OSS project will be classified as peripheral as the periphery includes the roles *active developers*, *peripheral developers*, *bug fixers* and *bug reporters* [13, 8].

2.3.3 Classification Metrics

There are different approaches to classifying developers according to the core/periphery dichotomy. All metrics aim to quantify the amount of participation a developer has in the project. One approach is based on *counting participation metrics*, e.g., lines of codes contributed or commits made. Although this method is widely used, it has been shown that count-based metrics cannot reflect more complicated structures and do not necessarily agree with developer perception [16].

Another approach of quantifying the developer's amount of participation relies on a *network-based evaluation.* Different measures of centrality can be employed to quantify the developer's structural importance or prominence in the network. Centrality can be equated to the contribution the vertex makes to the structure of the network [5, 4]. The simplest form of centrality is *degree-centrality* which is defined as the number of vertices a given vertex is linked to [4].

The measure of centrality used in this work is *eigenvector centrality* which characterises a vertice's centrality not only via the number of their links but also reflects the centrality of the nodes they are linked to [4]. Hence, a high eigenvector centrality can reflect not only that a developer coordinates with *many* developers but also that he/she is connected to other *important* project members [16].

After quantifying all developers' amounts of participation either via count-based or via network-based metrics, the classification proceeds as follows: A developer is classified as member of the core group if his/her level of participation is in the upper 20th percentile [17]. Thus, the core group is made up of those most active developers who are responsible for 80% of the project activity together. All other developers are classified as peripheral members [16].

As classifications based on the network approach have been shown to accurately reflect developer perception [16], we use eigenvector centrality on the source code-based developer network (see Section 2.2.2 for the construction of the networks) to classify project members in core and periphery in this study. Additionally, we evaluated all our results using a commit-count classification as a sanity check. We give more details on the classification process in Secion 3.2.

2.4 Related Work

Considerable research has been done into the developer roles as explained in Section 2.3 as well as the communication and coordination of developers in OSS projects. We give an overview of this research in the following. Then, we explain what little research exists combining the notion of developer roles with an analysis of the project communication and situate this study with regard to the existing work.

2.4.1 Related Work regarding Developer Roles

Crowston et al. [8] define and evaluate three analysis approaches to identifying core members of OSS projects: (1) those individuals that are officially named as developers of the project, (2) those developers who are responsible for the bulk of the code contributions, and (3) those developers which form the dense, strongly connected core of the developer network. They analyse these operationalisations on bug-tracker data of 116 OSS projects. Although the three measures do not agree with each other, all of them suggested that only a small fraction of the project developers constitute the project core [8].

Ye and Kishida [33] study developer roles in the Gimp project, using the publicly available, pre-defined roles in this project. They analyse the number of code contributions as well as the number of emails sent to the mailing lists for the *core members*, *active developers*, *peripheral developers*, *bug fixers* and *bug reporters*. Ye and Kishida find that the most active participants on the mailing list were mostly those developers who were defined as core members or as active developers. In addition, the authors study the role advancement processes in Gimp as well as investigating the motivation of peripheral participants [33].

Similarly, Krogh et al. [19] study the role advancement processes in Freenet. Classifying developers solely based on whether they had access to the project's VCS repository instead of using a participative metric, e.g., lines of code contributed, they analyse how newcomers join the community and begin contributing code. They construct different phases of the typical joining process, finding that newcomers following a "joining script", i.e., adapting to a certain level and type of activity, as well as particular specialisation mechanisms has a favorable influence on their role advancement [19].

Jergensen et al. [15] examine developer role advancement in open source ecosystems – i.e., systems of OSS projects that share components, technologies and social norms – using the example of the ecosystem surrounding Gnome. In contrast to other researchers, they find little support for the traditional "onion model" of developer role transition (see Section 2.3.1) when applied to a single software project. The model is slightly more fitting when regarding the entirety of the software ecosystem. Nonetheless, classical assumptions regarding the "onion model" are disproven for the example of the Gnome ecosystem, most interestinlgy the assumption that prior experience (in the project or a project of the same ecosystem) would lead to a higher centrality of the developer's contribution. Thus, the authors find limitations regarding the applicability of the classical role advancement model [15].

Joblin et al. [17] conduct research concerning the developer network structure of 18 large open-source projects. They use source code-based networks which represent developer collaboration on code artifacts and classify the developers into core and periphery according to network-based metrics (see Section 2.3.3). Their findings indicate that the role stability of the core members is much greater than that of peripheral developers. Additionally, Joblin et al. show that even as the general developer community loses its hierarchical structure with the project growth, the core group remains hierarchically arranged. This indicates that the two developer classes exhibit not only different levels of participation but also possess structural and positional differences [17].

Similarly, Terceiro et al. [31] investigate how the core and periphery differ concerning their influence on the source code's structural complexity. In their study of 7 OSS web server projects, the authors find that core members – possibly because of their deeper understanding of the code – introduce less structural complexity and, in instances of complexity-reducing activities, such as refactoring, remove more structural complexity than peripheral developers. With these results, they demonstrate that the differences between core and periphery lie not only in their activity level and behaviour but also in the complexity of the code they contribute and, thus, illustrate the importance of a stable core to OSS software projects [31].

The related work concerning OSS developer roles demonstrates that – although not all software projects may fit all characteristics of common models of developer roles – there is a clear distinction of core and periphery in the developer communities with only a small fraction of developers in the role of core members. This contrast is apparent in the number of email and source-code contributions as well as the developers position in the network and the quality of their contributions.

2.4.2 Related Work regarding Developer Communication

One of the most important reasons for communication in OSS projects is knowledge sharing, which significantly affects the software project as a whole because it ensures that (1) users are enabled to use the software and adequate support is provided and that (2) developers gain a better understanding of the source code, thus, being able to make better contributions. This aspect of the communication in OSS projects has been studied extensively.

Hemetsberger and Reinhardt [12] as well as Lakhani and von Hippel [21] examine user-to-user support on the Kde and Apache mailing lists, respectively. Sowe et al. [30] investigate the role and identification of *knowledge brokers*, i.e., those individuals that help users to find the resources from which to extract the answers to their technical questions, in the example of three non-developer mailing lists of Debian. They find that knowledge brokers are connected across lists and perform a very important role in the project [30].

Regarding the knowledge sharing among developers, Lanzara and Morner [22] study how artifacts help developers to gain technical, organisational, and institutional knowledge of the project. Canfora et al. [6] as well as Jensen et al. [13] analyse the mailing list interactions of new project members. Canfora et al.'s findings indicate that mentoring is perceived as important by the "newbies" and mentors alike. Additionally, they develope and evaluate a tool for recommending suitable mentors [6]. Jensen et al. investigate mailing-list discussions initiated by "newbies" to determine whether potential contributors are welcomed by polite, timely responses. They find that 80% of "newbie" posts receive (mostly timely) replies, which is positively correlated with poster's future contribution [13]. Guzzi et al. [11] study the development mailing list of Lucene and find that – albeit the list's declared intent of being the medium for discussions between developers – core developers participate in less than 75% of the threads, implementation details only amount for a third of the threads, and the mailing list is no longer the primary means of project communication [11].

Several studies were performed concerning efficient coordination in OSS developer communities despite the large number of possible communication partners. Bird et al. [2] study how subcommunities spontaneously arise in e-mail social networks of six OSS projects. Their findings indicate that this phenomenon occurs most often concerning technical discussions. Additionally, a validation with VCS data indicates that the subcommunities are strongly connected to collaboration on source code [2]. Mockus et al. [27] find different mechanisms of minimising coordination requirements in the OSS projects Apache and Mozilla. Kuk [20] investigates strategic interaction and knowledge sharing in the Kde developer mailing list. His work indicates that developers make strategic choices of which threaded discussions to join and, thus, concentrate their participation on epistemic interactions [20].

2.4.3 Combining Developer Roles and Communication Analyses

Although there has been considerable research into communication on OSS mailing lists as well as the classification of OSS developers into core and periphery, the combination of these two remains mostly unstudied.

Scialdone et al. [29] perform a qualitative study concerning the content of emails sent to the mailing lists of two OSS instant messaging projects. Focusing on the group maintenance behaviours, e.g., politeness and emotional expressions, in decision-making episodes, they find that negative politeness tactics are one of the most important instruments in assuring the health of the developer community. Although the authors could not find significant, consistent differences between the group maintenance behaviour of core and periphery, their findings suggest that core members feel more comfortable expressing their sense of belonging in the group than peripheral developers. However, their results may not be representative of the entirety of OSS projects due to the similarity of their two casestudies and the limited number of decision episodes analysed [29].

Joblin et al. [16] evaluate different operationalisations of the core/periphery classification on email as well as VCS data. Their findings reveal that network-based classification metrics, e.g., degree centrality, agree more with developer perception than count-based measures. Additionally, they suggest that – although email classifications seems more reflective of the developer's perception – mail-based and code-based classifications are both suitable for performing the classification. They find that core members show higher positional stability than other developer groups. Moreover, their results indicate that – when analysing the same network that is used for the classification – intra-core edges exhibit the highest edge probability and peripheral developers are more likely to coordinate with core developers than with other members of the periphery [16]

Bird et al. [1] analyse email social networks of the Apache project. Studying the topology of these developer networks, they find indicators of the participation inequality thought to be characteristic for OSS projects as well as indicators that an individual's activity level positively correlates with the number of people replying to him/her. Additionally, they find that the number of messages sent is strongly correlated to the level of activity regarding source code and document changes and that developers, as opposed to email-only contributors, hold important positions in the email social network. Most interestingly, they found that developers who are influential in the email network – measured through social network measures such as in-degree, out-degree and betweenness – exhibit higher levels of source code change activity [1]. Although this does not correspond exactly to our notions/operationalisations of core and periphery as defined in Section 2.3.3, it might indicate that being part of the email network core is positively correlated to activity and, presumably, influence concerning the code.

This thesis extends the findings of the aforementioned work by providing an in-depth look at the developer classes' communication activity, their interaction, as well as studying the longevity and intensity of the different groups' communication. By classifying developers according to code-based metrics and analysing the email social networks, we evaluate how classification on VCS data is reflected in the email developer communities.

We examine whether Bird et al.'s [1] findings of a positive correlation between importance in the mail network and source code activity can be extended to a correlation of importance in the *code* network and *email* activity levels. Similar to Joblin et al. [16], we study the interaction of the different developer classes but analyse not only the frequency of their interaction but also characteristics, e.g., intensity, of the intra- and inter-class communication. Additionally, our study combines classifications on one data source – the VCS – with analyses of the mail data to provide a richer understanding of the developer roles.

In order to allow for a better integration of our results into previous findings, as well as an indicator that the chosen classification metrics fit the data and reflect developer perception, several casestudies used by Joblin et al. [16] as well as the ApacheHTTP project on which Bird et al. [1] base their study are examined in this work. We give more detailed information on the subject systems in Section 3.4.

3 Study Design

In this section, we present details on our study design. We start by explaining our research questions in Section 3.1. Thereafter, we describe the corresponding dependent and independent variables in Section 3.2 and present our hypotheses based on these operationalisations in Section 3.3. In Section 3.4, we discuss our choice of casestudies and disclose characteristics of these subject projects. Finally, we explain the implementation and execution of this study in Section 3.5.

3.1 Research Questions

Research has shown many differences between core and peripheral project members, e.g., structural complexity introduced in contributions [31] as well as positional stability [16], apart from mere differences in activity levels. Joblin et al. [16] have shown that – when evaluating the same data source as used for classification – core developers coordinate most often with other core developer and members of the periphery prefer core developers over other peripheral contributors as coordination partners. Thus, we expect differences with which groups the core and peripheral developers – as classified based on their code contributions – communicate primarily similar to the results of the aforementioned study. In addition, it seems probable that this communication is reflected in the composition of thread contributors, hence, we anticipate that most threads contain at least one person with technical expertise, i.e., one or more of the dominant code contributors.

Research Question 1 (RQ1) – *With developers from which group do core (respectively peripheral) developers tend to communicate primarily?*

Not only do we expect the preference for communication partners to be tied to their group affiliation, but we also expect the inter- and intra-class communication episodes to exhibit different characteristics. For this, we study the communication between pairs of developers. It seems likely that, due to their prolonged and consistent involvement in the project and the need to coordinate long-term contributions, core-core communication is more long-lived than other forms of communication. Another interesting aspect we explore is the intensity of communication. This gives rise to our second research question:

Research Question 2 (RQ2) – *How do core-core, core-peripheral, and peripheral-peripheral developer communication differ in longevity and intensity?*

As research has shown that developers' activity on the mailing list is positively correlated with their activity regarding code contributions [1], we expect source-code activity to be reflected in elevated levels of mailing-list activity, meaning that those developers that are classified as core based on source-code networks, i.e., the most active and important committers, are responsible for a majority of the activity on the mailing list. Additionally, as we examine development mailing lists, we deem it likely that developers (as opposed to mailing-list-only participants) contribute the bulk of the communication. Moreover, we want to study how this level of activity changes over the course of the release cycle as peripheral contributions might be concentrated in certain phases of the cycle, e.g., directly after releases.

Research Question 3 (RQ3) – *To what extent do the two developer groups contribute to the total communication activity? Are there any changes over the course of a release cycle?*

3.2 Variables

In this section, we present the independent and dependent variables of our study and explain how we operationalise them. We give an overview of all variables in our study in Table 3.1.

3.2.1 Independent Variables

The independent variables for our study are the choices that we make concerning the network construction as well as the developer classification and our identification of communication episodes.

For our *developer-developer relation*, we use either subsequent contributions to the same mailing-list thread or subsequent changes of the same source-code artifact (see also Section 2.2.2). We consider *function* as the abstraction level for artifacts, i.e., two developers in the source-code networks are connected if they work on the same function in the source code. This fine-granular information results in authentic social networks that agree with developer perception [16].

Our classification is performed on the source-code cochange networks. The classification metric used is *eigenvalue centrality* which reflects the developers' structural importance in the network of contributors (see Section 2.3.3). We use two types of classification granularity for different hypotheses: (1) On the *global* level, we create one clasification for the entire network, i.e., spanning the entire analysed time span. (2) For the *local* classification level, we split the available data into three-month windows and create cochange networks based on the split data. We use three-month activity ranges as research has shown that, beyond this window size, the developer community does not change significantly but some temporal details can be lost or obfuscated [16]. For each one of these activity ranges, a classification is created.

As the time ranges of the two data sources – mailing-list data and VCS history – may not match, we compute mail-based activity ranges into which we split both sources of data as the mailing-list data is the primary subject of this study. In some cases, the commit-activity may start after the first activity range or end before the last one. In these cases, we filter out all activity ranges which are *completely* outside the commit-activity time span, i.e., which begin and end before, respectively after, the commit time range. Due to this filtering, we may not have complete VCS-data for the first and last activity ranges. Thus, for these ranges, the classification may not match the data as well and may distort the findings slightly. Nonetheless, we prefer this approach to completely filtering out the activity ranges in question, as we do not want to limit our analysed time ranges this strongly. Additionally, for most projects, the missing classification data at the beginning of the first, respectively end of the last, range is negligible and, hence, should not significantly influence our findings.

As some core developers no longer provide (many) code contributions but mainly coordinate other contributors' work on the mailing list [17, 8], we include *mailing-list-core developers* (ML-core developers) in the classification. For this, we analyse the mailing list activity divided into three-month activity ranges. After retrieving the classifications for these windows, we categorise those developer as ML-core developers, who were classified as core members in at least 50 % of the activity ranges or for 12 activity ranges, i.e., 3 years. Consequently, we prevent temporary activity from distorting our results, but ensure that we consider those developers who provide intensive support on the mailing list over a prolonged period of time as core developers. These ML-core developers are, then, added to the core of the global classification as well as all ranges of the local classification and removed from the peripheral and unclassified developer groups.

When studying communication between pairs of developers, we use the algorithm shown in Algorithm 1 to identify communication episodes. For this, we define the independent variable *communication episode time window* as 7 days, $cEThreshold = 7$. Hence, each communication episode between two developers corresponds to a sequence of contacts between the two developers where the time between two subsequent contacts is never longer than $cEThreshold$. These communication episodes are not based on thread-level information but aggregate information of all contacts between these developers. We choose this approach as some pairs of developers may exchange knowledge over multiple threads and a thread-level construction of the communication episodes would obfuscate these complex communication patterns.

We choose a value of 7 days for the communication-episode time window based on statistics of the project communication. We analyse the *temporal distances* of the communication, i.e., the temporal differences between subsequent mails in a threadand observe that for all projects, at least 80 % of the project communication shows temporal distances of 20 hours or less. In addition, in all but three of the subject projects, 95 % of the communication is characterised by temporal distances of 150 hours or less – although this varied greatly from 40 hours for Jailhouse to 200 hours for U-Boot, Django, and PostgreSQL. Thus, as this corresponds to temporal distances regarding whole threads instead of contacts between pairs of developers, we chose 7 days – or about 170 hours – as the threshold of pair-wise communication episodes. This threshold indicates that we expect two developers who communicate to contact each other at least once per 7 days. If they communicate more seldomly, we do not consider their contact as one communication episode but as separate instances of communication.

Algorithm 1 : Identification of Communication Episodes between Classes D_1 and D_2

```
Input : An edge-list $E$ of all interactions between Classes $D_1$ and $D_2$,
        i.e., a list where each element corresponds to one edge in the
        communication network and contains $author1$, $author2$
        (lexicographycally greater than $author1$), and a $timestamp$ of
        the interaction
Output : A list $CE$ containing all communication episodes between $D_1$
         and $D_2$, in which each element corresponds to one
         communication episode and is a list of all interactions in the
         context of this communication episode
begin
    $pair.interactions.list \longleftarrow$ split $E$ according to $author1$ and $author2$,
      creating a list of pair-wise interactions for every communicating
      pair of authors
    $CE \longleftarrow$ empty list
    for $pair.interactions \in pair.interactions.list$ do
        $episode \longleftarrow \emptyset$
        for $i \longleftarrow 2$ to $length\ of\ pair.interactions$ do
            $prev.interaction \longleftarrow pair.interactions[i-1]$
            $episode \longleftarrow episode \cup prev.interaction$
            $interaction \longleftarrow pair.interactions[i]$
            $time.diff \longleftarrow$
              $interaction[timestamp] - prev.interaction[timestamp]$
            if $time.diff > 7\ days$ then
                add $episode$ to $CE$ as last element
                $episode \longleftarrow \emptyset$
        if $episode\ non\text{-}empty$ then
            add $episode$ to $CE$ as new element
```

3.2.2 *Dependent Variables*

Our dependent variables can be divided into three categories: variables pertaining to the class activity, variables with regards to the class interaction, and variables characterising communication episodes. All dependent variables will be introduced and explained in the following.

For simplicity we let $\mathit{Classes} = \{C, P, U, \mathit{total}\}$ denote the set of classes, i.e. core, periphery, unclassified, and total (all project members), respectively. Contributors are categorised as *unclassified* in case they only contribute to the mailing list, i.e., are not active in the VCS, and are not classified as ML-core developers. Let $D, D_1, D_2 \in \mathit{Classes}$, $n \in \mathbb{N}_0$, and $x \in [0;1]$ in the following.

Based on the classification, we obtain several general variables for a given communication network: $n(D)$ denotes the *number of D-members*, i.e., the absolute number of project members assigned to the class D by our chosen classification. Derived from this value, we define the *relative number of D-members* $rn(D) = n(D)/n(\mathit{total})$ as the proportion of contributors that are classified as D out of all project members.

Dependent Variables: Class Activity For a given email social network and corresponding developer classification, we characterise the developer classes' level of activity via the following variables. We let $\mathit{mails}(D)$ denote the *number of mails from D*, i.e., the absolute number of emails written by members assigned to the class D. Analogously, we define the *number of threads started by D* as $\mathit{threadsStarted}(D)$. This variable corresponds to the absolute number of threads where the initial email was written by a contributor classified as D – at the time the initial mail was sent in case local classifications are used. We only consider threads that started a communication, i.e., theads that received at least one answer, as unanswered threads could indicate either auto-generated messages, e.g., emails from the continuous build system, or spam [11]. Messages sent to the wrong mailing list, e.g., support questions posted to a development mailing list, are a third possibility and should also be filtered out as they do not provide valuable insights into the communication patterns on the primary *development* mailing list.

We derive several variables from these absolute numbers. First, we introduce the *average number of mails by D*, $\mathit{aMails}(D) = \mathit{mails}(D)/n(D)$, and the *average number of threads started by D*, $\mathit{aThreadsStarted}(D) = \mathit{threadsStarted}(D)/n(D)$, which normalise the activity on the basis of single developers. In addition, we define measures of the activity of the entire developer class. For this, we let $\mathit{rMails}(D) = \mathit{mails}(D)/\mathit{mails}(\mathit{total})$ denote the *relative number of mails written by D*, i.e., the proportion of mails that are written by participants assigned to class D out of all mails. We define the *relative number of threads started by D* analogously as $\mathit{rThreadsStarted}(D) = \mathit{threadsStarted}(D)/\mathit{threadsStarted}(\mathit{total})$. Both of these variables are indicators of how much each class contributes to the total communication activity in the project.

Dependent Variables: Class Interaction For the analysis of the classes' interaction, we define the following dependent variables for a given communication network and classification. We analyse the edges of the network as well as the threaded mailing list discussions, filtering out email threads without responses as explained above.

Let $links(D_1, D_2)$ denote the *number of links from* D_1 *to* D_2, i.e., the absolute number of edges in the network whose origin is a contributor classified as D_1 and whose destination is a member of group D_2. From this variable, we derive the *proportion of links from* D_1 *to* D_2, $rLinks(D_1, D_2) = links(D_1, D_2)/links(D_1, total)$, which corresponds to the proportion of links from contributor class D_1 to class D_2 out of all links originating from D_1.

Moreover, we let $threads(D, n)$ denote the *number of threads with maximum n D-contributors*. This value corresponds to the absolute number of threads where the number of participants assigned to class D is less than or equal to n. Hence, it is a sum of the number of threads with exactly m D-contributors for $m \in \{0, \dots, n\}$. When using local classifications, it is possible – although seldom – that a thread spans multiple classification ranges (see Figure A.1). In these cases, we regard all those developers as core who are classified as core in at least one of the ranges as it is likely that developers that are categorised as core in one range already held similar positions in the previous and following range. As shown in Figure A.2, an exceedingly small number of developers is ever re-classified due to this definition, hence, this assumption should not distort our results. Similarly, we regard every developer as peripheral who was part of the periphery in at least one of the thread's ranges and is classified as core in none of these time spans.

From this, we derive $rThreads(D, n) = threads(D, n)/threads(total, \infty)$ where $thread(total, \infty)$ equals the total number of threads, as it represents the number of threads where the number of participants is $\leq \infty$. Thus, the variable $rThreads(D, n)$ represents the *proportion of threads with maximum n D-contributors*, i.e., the proportion of threads where the number of participants assigned to class D is less than or equal to n out of all threads.

Dependent Variables: Comunication Episodes When studying communication between pairs of developers, we use the Algorithm 1 as described in Section 3.2.1 to identify communication episodes. We let the term $commEp(D_1, D_2)$ denote the set of communication episodes between pairs of contributors where one is assigned to class D_1 and the other is classified as D_2. In some of these communication episodes, only one contact may occur. These communications need to be filtered out for some metrics. Thus, we let $commEpMulti(D_1, D_2) \subseteq commEp(D_1, D_2)$ be the set of communication episodes between D_1 and D_2 in which only episodes with multiple interactions are included. These multiple contacts do not necessarily correspond to two-way communication, i.e., answers, but may indicate that one developer has contacted the other developer multiple times without receiving an answer.

For a given communication episode, e, we let $nrLinks(e)$ and $nrMails(e)$ denote the *number of edges* produced and the *number of e-mails* written between the two developers in the context of e, respectively. The number of edges may not necessarily equal the number of mails – although this is sometimes the case – as one mail can produce multiple edges due to the network construction algorithm we employ (see Section 2.2.2). In addition, we define the *duration* in days as $dur(e)$. Based on these basic characteristics, we derive the *intensity* of the communication episode as $intensity(e) = nrLinks(e)/\lceil dur(e) \rceil$ that is the number of links per day. We count each day that is started as a whole day, e.g., considering a 2-hour communication episode as 1 day, in this calculation. This prevents very short but intensive communication episodes, e.g., 5 emails in 30 minutes, from skewing the distribution of the communication intensity. Analogously, we derive the communication intensity measured in mails per day, *mail-based intensity*, as $mailIntensity(e) = nrMails(e)/\lceil dur(e) \rceil$. Both these measures of intensity are only defined for communication episodes with multiple contacts, as we cannot sensibly define an intensity for just one contact.

For our study, we investigate the *class-level aggregation* of thesecharacteristics. Thus, we define the *class communication intensity between D_1 and D_2* as $cIntensity(D_1, D_2) = \bigcup_{e \in commEpMulti(D_1,D_2)} intensity(e)$. Thus, $cIntensity(D_1, D_2)$ corresponds to a (multi-)set which contains the communication intensities for all communication episodes between pairs of developers assigned to D_1 and D_2, respectively. Analogously, we let the variable $cMailIntensity(D_1, D_2) = \bigcup_{e \in commEpMulti(D_1,D_2)} mailIntensity(e)$ denote the *class communication mail-based intensity between D_1 and D_2*. Similarly, we let $cLongevity(D_1, D_2) = \bigcup_{e \in commEp(D_1,D_2)} dur(e)$ denote the *class communication longevity between D_1 and D_2*. Additionally, we define the *class communication longevity of multi-contact episodes between D_1 and D_2* via $cLongevityMulti(D_1, D_2) = \bigcup_{e \in commEpMulti(D_1,D_2)} dur(e)$.

When working with the class communication intensities and longevities, we want to compare the observations for different class combinations, e.g., to establish whether core-core communication is of greater longevity than core-peripheral communication. For this, *quantiles* as defined in the following are well suited. Quantiles (or, similarly, *percentiles*) essentially divide a given ordered data into subsets of sizes corresponding to given probabilities, such that the first subset contains the smallest values and the last subset contains the highest values. Mathematically, when given a set of n observations arranged in order of magnitude, the q-th quantile is the value that exceeds q% of the measurements and is less than the remaining $(1-p)$% [26]. Thus, if q is the 0.1-sample quantile, it corresponds to the threshold under which 10% of the data values lie. Illustratively, when picking a random value out of the dataset it will be $\leq q$ with a probability of 10 %. Probabilities of 0 and 1 correspond to the smallest and largest value of the data, respectively [28].

In our work we use quantiles as follows: $q(data, x)$ corresponds to the x-th quantile of the given *data*. Thus, we represent the x-th quantile of the class communication intensity between D_1 and D_2 and of its longevity via the terms $q(cIntensity(D_1, D_2), x)$ and $q(cLongevity(D_1, D_2), x)$, respectively.

Additionally, we use the *variance* to quantify how variable a given dataset is, i.e., illustratively, how far apart the higher and lower values of the data are. Mathematically, the variance of a sample of n measurements corresponds to the squared deviations of the measurements from their mean divided by $(n-1)$, which equals the short form $\frac{(\sum x_i^2) - (\sum x_i)^2/n}{n-1}$ [26]. The interested reader is referred to Mendenhall et al. [26] for more details on the variance as well as quantiles.

We let $var(data)$ denote the variance of *data*. Thus, we use the terms $var(cIntensity(D_1, D_2)$ and $var(cLongevity(D_1, D_2)$ for the variance of the class communication intensity and longevity, respectively, between D_1 and D_2.

Tab. 3.1 Independent and Dependent Variables of our Study along with their Corresponding Description

Variable	**Description**
	Independent Variables
Developer-developer relation	A contributors posting mails to a thread constitutes an interaction with all previous participants in the thread.
Classification	Developers are classified as core or periphery on the basis of eigenvector centrality on function-level code-cochange networks constructed based on the VCS data. Longterm-mailing list core contributors are included as core members. For the local classification, we consider 3-month time windows.
Communication episode time-window $cEThreshold$ =7 days	An interaction between a pair of developers belongs to a communication episode in case it occurs no more than 7 days later than the most recent interaction in the episode.
	Dependent Variables: General
Number of D-contributors $n(D)$	Absolute number of developers classified as D for a given network and classification
Relative number of D-contributors $rn(D) = n(D)/n(total)$	Proportion of developers classified as D for a given network and classification based on the total number of developers
	Dependent Variables: Developer Class Activity
Number of mails from D $mails(D)$	Absolute number of mails sent by members of D

(Continued) Independent and Dependent Variables of our Study along with their Corresponding Descriptions

Variable	**Description**
Number of threads started by D $threadsStarted(D)$	Absolute number of threads initiated by members of D
Average number of mails by D-members $aMails(D) = mails(D)/n(D)$	Average number of mails sent by a member of class D
Average number of threads started by D-members $aThreadsStarted(D) = threadsStarted(D)/n(D)$	Average number of threads initiated by a member of class D
Relative number of mails written by D $rMails(D) = mails(D)/mails(total)$	Proportion of mails written by members of class D out of all mails
Relative number of threads started by D $rThreadsStarted(D) = threadsStarted(D)/threadsStarted(total)$	Proportion of threads initiated by members of class D out of all threads
Dependent Variables: Developer Class Interaction	
Number of links from D_1 to D_2 $links(D_1, D_2)$	Absolute number of links sent from a member of class D_1 to a member of class D_2
Proportion of links from D_1 to D_2 $rLinks(D_1, D_2) = links(D_1, D_2)/links(D_1, total)$	Proportion of links sent from a member of class D_1 to a member of class D_2 out of all links emanating from D_1
Number of threads with maximum n D-contributors $threads(D, n)$	Absolute number of threads in which n or fewer members of class D participate

(Continued) Independent and Dependent Variables of our Study along with their Corresponding Descriptions

Variable	**Description**
Proportion of threads with maximum n D-contributors $rThreads(D,n) = threads(D,n)/threads(total,\infty)$	Proportion of threads in which n or fewer members of class D participate out of all threads
Dependent Variables: Communication Episodes	
Class communication longevity between D_1 and D_2 $cLongevity(D_1,D_2) = \bigcup_{e \in commEp(D_1,D_2)} dur(e)$	(Multi-)set containing the duration (in days) of all communication episodes between pairs of developer where one is member of D_1 and the other member of D_2
Class communication longevity (multiple contact episodes) between D_1 and D_2 $cLongevityMulti(D_1,D_2) = \bigcup_{e \in commEpMulti(D_1,D_2)} dur(e)$	(Multi-)set containing the duration (in days) of communication episodes between pairs of developer where one is member of D_1 and the other member of D_2, considering only communication episodes with multiple interactions
Class communication link-intensity between D_1 and D_2 $cIntensity(D_1,D_2) = \bigcup_{e \in commEpMulti(D_1,D_2)} intensity(e)$	(Multi-)set containing the intensity (in links/day) of all communication episodes between pairs of developer where one is member of D_1 and the other member of D_2, considering only episodes with multiple interactions
Class communication mail-intensity between D_1 and D_2 $cMailIntensity(D_1,D_2) = \bigcup_{e \in commEpMulti(D_1,D_2)} mailIntensity(e)$	(Multi-)set containing the intensity (in mails/day) of communication episodes between pairs of developer where one is member of D_1 and the other member of D_2, considering only episodes with multiple interactions

3.3 Hypotheses

Prior research has shown that core developers take on the role of coordinating other developers' work [8]. Hence, it seems likely that members of the periphery will communicate extensively with core developers as peripheral contributors typically do not have comprehensive knowledge of the project [17] and, thus, need to be advised as to how to integrate their contributions. Additionally, as prior research has indicated [16], we expect core developers to coordinate primarily with other core members.

Hypothesis 1.1 (H1.1) – *Peripheral developers as well as core developers will prefer core developers as communication partners..*
We expect $rLinks(P, C) > rn(C)$ *and* $rLinks(C, C) > rn(C)$ *to hold.*

Core developers characteristically possess extensive project knowledge [17] and one of the development mailing list's goals is the sharing of knowledge between developers. Additionally, we expect high levels of code contribution activity to be reflected in high levels of mailing list participation. Therefore, we anticipate that core members are present in most threaded mailing list discussions.

Hypothesis 1.2 (H1.2) – *In a majority of e-mail threads, at least one core developer will participate.*
We expect $rThreads(C, 0) < 50\%$ *to hold.*

Members of the periphery typically do not know as much about the system and do not need to coordinate other developers' efforts [8]. Thus, they won't be able to contribute as much knowledge to threads and it is unlikely that many peripheral developers communicate in one threaded discussion. We expect peripheral contributors' main role to be asking questions that are answered by knowledgeable core members instead of other peripheral developers contrary to them providing advice and opinions. Hence, we formulate the following hypothesis:

Hypothesis 1.3 (H1.3) – *In a typical e-mail thread, no more than three different peripheral developers will participate.*
We expect $rThreads(P, 3) > 50\%$.

We expect core-peripheral communication episodes to be rather short-lived as the core developer will typically coordinate one of the irregular contributions of the peripheral developer. As the latter might need intensive advising on the task or go through an iterative process of refining his/her contribution according to the core developer's advice before it is accepted into the project's code base, the core-peripheral communication episodes may exhibit high intensities.

Hypothesis 2.1 (H2.1) – *Communication between a core developer and a peripheral developer will tend to be short-lived, but there may be intensive communication over this short time-span.*

We anticipate $q(cLongevity(C, P), 0.5) < q(cLongevity(total, total), 0.5)$ *and* $q(cIntensity(C, P), 0.75) > q(cIntensity(total, total), 0.75)$.

When two core developers communicate, this might be an indicator that they generally work on overlapping parts of the project. As they are characterised by long-term involvement [16] and may have rather fixed areas of expertise in the project, they might need to communicate over a longer time period in order to coordinate aspects of the evolving system architecture.

Some communication episodes may exhibit high intensities, e.g., project coordination when planning future releases or restructuring parts of the system architecture, whereas others could show very low intensity levels, e.g., two core developers working on loosely connected parts of the project who need to coordinate only occasionally over a longer time period.

Hypothesis 2.2 (H2.2) – *Communication between two core developers will show greater longevity, but may vary in intensity.*
We expect $q(cLongevity(C, C), 0.5) > q(cLongevity(total, total), 0.5)$ *as well as* $var(cIntensity(C, C)) > var(cIntensity(total, total)$.

Peripheral developers do not have extensive project knowledge [17] and can, thus, only give little advice to other peripheral contributors and have discussions of limited depth with members of their developer class. Moreover, they do not have to coordinate larger efforts or matters of the system architecture, instead probably communicating primarily when they coincidentally work on the same part(s) of the system. Another possibility of peripheral-peripheral communication episodes would be that both developers make smaller contributions to a general discussion, e.g., feature suggestions in a discussion of future plans for the project. This gives rise to our hypothesis that:

Hypothesis 2.3 (H2.3) – *Communication between two peripheral developers will be rather short-lived.*
We expect $q(cLongevity(P, P), 0.5) < q(cLongevity(total, total), 0.5)$ *to hold.*

Similar to the findings of Bird et al. [1], we expect activity with regards to the OSS project's code base to be reflected in the distribution of the mailing list traffic.

Hypothesis 3.1 (H3.1) – *Code contributors will be more active regarding the number of mails written than mailing-list-only contributors.*
We anticipate $aMails(C) > aMails(U)$ *as well as* $aMails(P) > aMails(U)$.

As core developers take on roles as coordinators as well as form the project leadership structure [16, 8] and as prior research has shown that activity on the mailing list is positively correlated with activity measured via code contributions [1], we expect the project core to be dominant with regards to the mailing list traffic, in particular the number of active threads started.

Hypothesis 3.2 (H3.2) – *Core developers will write a majority of emails and be responsible for starting more than 50% of mailing list threads.*
We expect $rMails(C) > 50\%$ *and* $rThreads(C) > 50\%$ *to hold.*

We expect the developers' level of activity to change over the course of a release cycle. In particular, we expect the contributions of core members – the project leadership structure [17] – to increase before major releases as their coordination effort most likely intensifies in the context of discussions of feature freezes and determining details of the releases.

For the evaluation of our release-specific hypotheses, quantitative analyses are too error-prone due to temporary fluctuations in activity, difficulties in discerning major and minor releases due to changing release patterns in some systems, and – most importantly – overlapping release cycles, e.g., v4.2 can already be in preparation before the release of v4.1.9. These problems cannot be adequately addressed with quantitative measures and, thus, we choose to use qualitative methods to evaluate the following, release-specific hypotheses.

Hypothesis 3.3 (H3.3) – *The number of e-mails written by core developers will increase significantly in the time period before a release.*

We expect releases to draw the attention of the more passive developers as well as the user group, provoking some of them to become (more) active contributors. As explained above, we use a qualitative analysis to evaluate this hypothesis.

Hypothesis 3.4 (H3.4) – *The communication activity of peripheral developers will spike during the beginning and end phase of a release cycle.*

We expect the intensified need for coordination prior to releases to increase not only the number of mails written by core developers, but particularly the communication among project core. We use qualitative measures to assess this hypothesis due to the inaccuracy of quantitative measures.

Hypothesis 3.5 (H3.5) – *Core-core developer communication will increase in the time period prior to a release*

3.4 Casestudies

For our empirical study of the hypotheses introduced in Section 3.3, we analyse 11 OSS projects listed in Table 3.2. We choose the subject projects in such a way as to ensure diversity and avoid biasing our results. The chosen casestudies differ in the following dimensions: (1) Application domain: We investigate projects that belong in the user and developer domain as well as operating systems. (2) Technology: There is considerable diversity in the programming languages and libraries used in the projects. (3) Source-code activity: Our projects range from circa 20 to 1 500 source-code contributors and about 3 800 to 480 000 commits in the studied time range. (4) Mailing-list activity: The subject systems vary greatly in mailing-list activity, ranging from about 120 to more than 4.600 ML-contributors with circa 3.200 to 480.000 mails posted to the primary development mailing list. (5) Communication guidelines: We ensure that some of the projects, e.g., Qemu, follow very traditional OSS communication guidelines meaning that the mailing list is the hub of project communication. Other projects, such as Django, present a more modern form of OSS communication, relying on features of the hosting platform or other means of communication, e.g., issue trackers. (6) VCS: Although we extract our data based on the projects' Git repositories or mirrors, the casestudies differ in the version control system used for development – most use Git, but we also investigate several projects that rely on Svn as VCS as well as Llvm which uses Subversion. (7) Release system: Some systems, e.g., Chromium, are characterised by a very regular, rigid release schedule, whereas others, e.g., Jailhouse, publish releases relatively irregularly and without an explicit system detailing the phases of the release cycle.

For each project, we analysed the activity on the primary development mailing list as well as the source-code contribution for a time span of at least 1 year. This prevents our findings from being too strongly influenced by temporary tendencies. The projects chosen include ApacheHTTP which Bird et al. [1] analysed as well as several projects studied by Joblin et al. [16](see Section 2.4). This ensures that the classification metric chosen yields reasonable core and periphery classifications which agree with developer perception [16]. Additionally, it allows us to extend and validate the findings of the aforementioned studies.

3.5 Implementation and Execution

We implement all calculations and analyses in self-written scripts in the statistical programming language R[1] [28]. The source code of this study will be integrated into the repository `se-passau/dev-network-growth` which will be made public in the future[2]. The raw data as well as the constructed author networks for all subject projects are based on the data extracted via the tool Codeface[3].

[1] `https://www.r-project.org/`

[2] The complete source code is enclosed on the digital copy of this thesis.

[3] `https://github.com/siemens/codeface`

Tab. 3.2 Overview of Subject Projects

Project	Domain	Language	#Dev	#M-Dev	Time Range	#Mails	#Commits
ApacheHTTP	Devel	C	82	2 112	03/2002-06/2016	54 921	39 735
Busybox	User	C	221	2 701	01/2003-02/2016	42 013	43 682
Chromium	User	C/++, JS	1 457	1 237	12/2014-12/2015	8 576	479 064
Django	Devel	Python	318	457	12/2014-12/2015	3 257	13 878
Ffmpeg	User	C	201	443	01/2015-02/2016	19 508	13 685
Jailhouse	Devel	C	17	123	11/2013-08/2016	5 619	3 836
Llvm	Devel	C/++	272	987	12/2014-12/2015	11 119	26 613
OpenSSL	Devel	C, Perl	165	4 677	09/2001-02/2016	32 659	38 473
Qemu	OS	C	951	7 131	04/2003-07/2016	430 561	196 319
U-Boot	Devel	C	281	703	12/2014-12/2015	37 595	13 791
Wine	User	C	94	196	01/2015-01/2016	4 871	22 967

4

Results

In the following, we present our findings for all hypotheses defined in Section 3.3. We only present the results using eigencentrality as classification metric and, for each hypothesis, we give an overview of the findings for all projects and go into more detail for a few typical or notable suject projects. More detailed findings, additional plots, and all results using the commit-count classification metric can be found on the digital copy of this work.

4.1 Preferences concerning Communication Partners (H1.1)

In **H1.1**, we formulated our expectation that core developers as well as peripheral developers would prefer core developers as communication partners. We anticipated $rLinks(C, C) > rn(C)$ and $rLinks(P, C) > rn(C)$, i.e., that edges to core developers would appear more often than we would expect based on mere developer class frequency without preferences for any developer class. In Table 4.1, we show the relative developer class frequencies for core, peripheral, and unclassified, as well as the relative link frequencies between the three classes for all our subject projects. This corresponds to the aggregated data of all activity ranges.

Our first observation relates to the developer class frequencies: As prior research has shown [16], core developers make up only a very small part of the developer community, ranging from about 5 % in OpenSSL to circa 25 % in Wine. Interestingly, although the mail data we analyse stems from the primary development mailing list, the proportion of non-developers, i.e., mailing-list-only contributors, is quite high, ranging from about 40 % in Wine to more than 90 % in OpenSSL. In all but two cases, unclassified project members make up more than half of the mailing list contributors. The proportion of peripheral developers shows strong differences between the subject projects, varying between an almost negligible 3 % in OpenSSL to about 40 % in Ffmpeg.

Additionally, our findings show that all contributor classes have a clear preference for core members when sending out ties, as $rLinks(C, C) > rn(C)$, $rLinks(P, C) > rn(C)$, and $rLinks(U, C) > rn(C)$ hold for all casestudies. How pronounced this difference is varies significantly. We define a *preference factor of D_1 by D_2* as $pref(D_1, D_2) = rLinks(D_2, D_1)/rn(D_1)$ which quantifies how much D_2-contributors prefer members of D_1 compared to random communication partners. For instance, a communication factor $pref(C, P) = 2$ would indicate that peripheral members are twice as likely to communicate with a core member than with a random project participatn. The preference factors for all developer class combinations are shown in Table 4.2.

In Wine, core members are only preferred to other members by a factor of 2.50, 2.61, and 2.39 by core, peripheral, and unclassified project members, respectively. The subject project Qemu is in strong contrast to this, as core members are more likely to communicate with another core member as opposed to a random contributor by factor 9.12. Peripheral and unclassified members of this project also strongly prefer core members – by factors 9.06 and 7.01 respectively. Except for Wine, all contributor classes in all casestudies prefer core by a factor of at least 3.

There does not seem to be any overarching tendency that this preference is more pronounced in one participant class. While the preference of core members as communication partners is slightly stronger in developers than in ML-only contributors in the projects Jailhouse, Llvm, Qemu, and Wine, the opposite effect is visible in Busybox and Ffmpeg. These values seem like insignificant, expectable fluctuations and indicate that the strength of the preference for core contributors is a project-wide tendency which transcends member classes.

Examining the preference factor of unclassified members, we observe that no clases in any of our subject projects prefer unclassified members. In fact, the preference factor is nearly always smaller than 0.5, indicating that members are nearly twice as likely to communicate with a random project member than with a random unclassified contributor. There are no notable differences in the distinctness of this preference in different classes.

When investigating possible preferences of peripheral developers as communication partners, the findings are less clear. Although there is a negative preference of peripheral members in all developer classes in ApacheHTTP, Chromium, Ffmpeg, Jailhouse, Llvm, and Wine, we observe a considerable class-invariant preference of peripheral contributors in OpenSSL. For this casestudy, the preference factor of peripheral members by peripheral members is greater than 6.5. In addition, the core and peripheral developers in Busybox and Qemu seem to slightly prefer peripheral developers as communication partners.

These aggregated results with regards to developer class frequencies, link frequencies and communication partner preferences stay fairly consistent over time. We go into more detail regarding this evolutionary perspective in Section 4.10. The aggregated data for the project Ffmpeg as shown in Table 4.1 are also reflected in Figure 4.1 which shows the evolution of the link frequencies. We observe that, although there are slight fluctuations, all three classes exhibit a clear and consistent preference for core members as communication partners. Thus, we draw the conclusion that this preference for core members is not only class-invariant but also very consistent over time. In addition, we observe that – in this project – the preference of peripheral members to unclassified members is most pronounced in the core developer group where it also exhibits relative consistency. In the other two developer groups, the relative link frequencies to unclassified and peripheral developers show considerably stronger fluctuations although peripheral members are slightly preferred.

In conclusion, we **accept H1.1** as a clear preference for core members as communication partners exists in all participant classes in all casestudies.

4.2 Core Thread Participation (H1.2)

In **H1.2**, we formulate our expectation that, in a majority of threads, at least one core developer will participate. In Table 4.3, we show $rThreads(C, n)$ for $n \in \{0, \dots, 9\}$ for all subject projects. As described in Section 3.2, this corresponds to the proportion of threads where maximum n core developers contribute and, thus, we expect $rThreads(C, 0) < 50\%$.

We observe, that this is the case for all eleven subject projects, i.e., there are only few threads without core member participation, although the degree of this core involvement varies. In Ffmpeg, Jailhouse, and Wine, there are almost no threads without core members – the proportion of threads without core participation is smaller than 10 %. OpenSSL differs greatly from these three casestudies as its proportion of threads without core members is about 47 %. In most projects, about 10 % to 30 % of the threaded mailing list discussions are without core participation.

Interestingly, it seems as though it happens rather seldomly that multiple core developers participate in one thread: In all projects except Ffmpeg and Wine, we observe that most threads show maximum 1 core contributors. In OpenSSL, more than 90 % of all threads are characterised by 1 or 0 core contributors. Additionally, we observe for all projects that in more than three quarters of threads maximum 2 core developer contribute. Undoubtedly, there are some threads with very high numbers of core participants, e.g., 0.18 % of threads show 17 core developers in Llvm, but these numbers seem like outliers and we observe a general tendency in all projects towards a very small number of threads without core participation and a high proportion of threads with only one core contributors.

As shown above, core developers contribute to a majority of threads in all projects. Additionally, in all but one project less than a third of threads are without core participation. Thus, we **accept H1.2**.

4.3 Peripheral and Unclassified Thread Participation (H1.3)

In **H1.3**, we hypothesise that, in a typical mailing-list thread, no more than three different peripheral developers will participate, i.e., we anticipate $rThreads(P, 3) > 50\%$. We present our findings of $rThreads(P, n), n \in \{0, \dots, 8\}$ in Table 4.4. In addition, we show the analogous results regarding non-developer involvement, i,e, $rThreads(U, n)$, in Table 4.5.

Tab. 4.1 Relative Developer Class Frequencies and Relative Link Frequencies between Core and Periphery

Casestudy	$rn(D)$ [%]			$rLinks(D_1, D_2)$ [%]								
	Core	Periph.	Unclass.	(C,C)	(C,P)	(C,U)	(P,C)	(P,P)	(P,U)	(U,C)	(U,P)	(U,U)
ApacheHTTP	17.28	8.17	74.56	61.47	6.14	32.39	61.83	7.6	30.57	61.64	6.21	32.15
Busybox	10.78	6.5	82.72	49.24	7.19	43.57	52.97	8.5	38.52	54.83	3.87	41.29
Chromium	10.68	35.29	54.03	54.07	22.92	23.01	50.22	26.2	23.58	51.69	24.01	24.29
Django	12.33	9.47	78.2	49.45	11.42	39.12	62.91	1.58	35.51	50.81	8.48	40.71
FFmpeg	15	39.14	45.86	76.01	17.27	6.72	74.07	20.36	5.57	79.16	14.4	6.44
Jailhouse	17.42	9.55	73.03	71.54	6.9	21.56	67.79	6.22	25.99	66.26	7.72	26.02
Llvm	12.71	17.19	70.1	61.64	15.17	23.19	59.04	16.79	24.17	57.11	14.01	28.89
OpenSSL	5.6	2.66	91.74	17.99	9.41	72.59	27.88	17.58	54.55	25.66	6.96	67.38
Qemu	6.96	21.38	71.66	63.5	25.85	10.65	63.03	26.82	10.15	48.79	18.38	32.83
U-Boot	9.13	37.96	52.91	73.77	19.85	6.38	48.11	41.21	10.68	61.81	21.34	16.85
Wine	25.39	36.39	38.22	63.58	16.44	19.99	66.19	21.58	12.23	60.66	9.92	29.42

Tab. 4.2 Preference Factor for Communication Partner Classes

Casestudy	$pref(D_1, D_2) = rLinks(D_2, D_1)/rn(D_1)$								
	Pref. Factor for Core			Pref. Factor for Periph.			Pref. Factor for Unclass.		
	(C,C)	(C,P)	(C,U)	(P,C)	(P,P)	(P,U)	(U,C)	(U,P)	(U,U)
ApacheHTTP	3.56	3.58	3.57	0.75	0.93	0.76	0.43	0.41	0.43
Busybox	4.57	4.91	5.09	1.11	1.31	0.60	0.53	0.47	0.50
Chromium	5.06	4.70	4.84	0.65	0.74	0.68	0.43	0.44	0.45
Django	4.01	5.10	4.12	1.21	0.17	0.90	0.50	0.33	0.52
FFmpeg	5.07	4.94	5.28	0.44	0.52	0.37	0.15	0.12	0.14
Jailhouse	4.11	3.89	3.80	0.72	0.65	0.81	0.30	0.36	0.36
Llvm	4.85	4.65	4.49	0.88	0.98	0.82	0.33	0.34	0.41
OpenSSL	3.21	4.98	4.58	3.54	6.61	2.62	0.79	0.59	0.73
Qemu	9.12	9.06	7.01	1.21	1.25	0.86	0.15	0.14	0.46
U-Boot	8.08	5.27	6.77	0.52	1.09	0.56	0.12	0.20	0.32
Wine	2.50	2.61	2.39	0.45	0.59	0.27	0.52	0.32	0.77

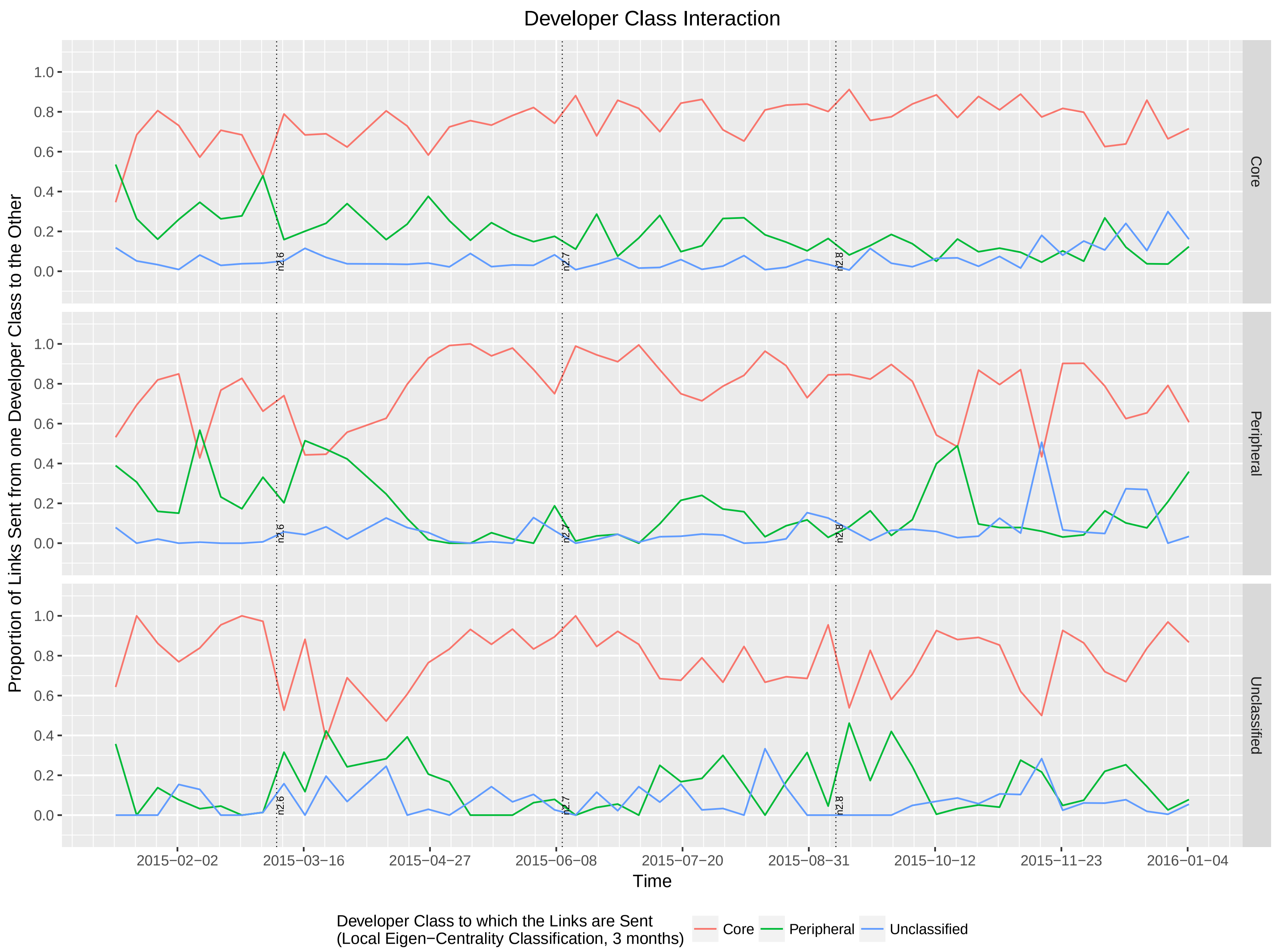

Fig. 4.1 Relative Edge Class Frequency Evolution in Ffmpeg

Tab. 4.3 Proportion of Threads with Maximum n Core Developers: $rThreads(C, n)$ [%]

Casestudy	n									
	0	1	2	3	4	5	6	7	8	9
ApacheHTTP	14.8	51.8	81.39	91.77	96.04	97.96	99.02	99.48	99.76	99.88
Busybox	18.35	63.74	88.51	96.23	98.66	99.57	99.96	99.99	100	100
Chromium	30.15	69.06	84.92	92.82	95.62	97.34	97.99	98.64	98.92	99.14
Django	10.4	51.3	76.12	89.6	94.56	96.93	98.82	99.29	99.76	100
FFmpeg	2.93	32.47	77.21	90.82	96.22	98.47	99.19	99.72	99.91	100
Jailhouse	5.29	51.98	94.05	100	100	100	100	100	100	100
Llvm	28.25	68.58	87.07	92.92	95.06	97.19	98.05	98.66	99.21	99.45
OpenSSL	47.35	91.29	98.75	99.74	99.94	99.98	100	100	100	100
Qemu	16.7	52.56	81.46	92.96	97.12	98.76	99.44	99.76	99.86	99.92
U-Boot	19.01	55.61	90.5	97.73	99.36	99.77	99.94	100	100	100
Wine	8.41	46.38	89.63	97.16	98.63	99.51	99.8	99.8	99.9	99.9

Tab. 4.4 Proportion of Threads with Maximum n Peripheral Developers: $rThreads(P, n)$ [%]

Casestudy	n								
	0	1	2	3	4	5	6	7	8
ApacheHTTP	79.76	97.23	99.57	99.9	99.98	100	100	100	100
Busybox	85.96	99.08	99.92	99.99	100	100	100	100	100
Chromium	37.19	74.08	89.52	96.05	98.71	99.35	99.64	99.78	99.93
Django	76.6	97.4	99.76	100	100	100	100	100	100
FFmpeg	66.03	94.54	99.28	99.78	99.94	100	100	100	100
Jailhouse	76.87	98.02	100	100	100	100	100	100	100
Llvm	59.85	89.99	97.38	98.9	99.57	99.88	99.88	99.88	100
OpenSSL	89.06	98.61	99.76	99.92	99.96	100	100	100	100
Qemu	51.87	87.85	97.84	99.59	99.92	99.97	99.99	100	100
U-Boot	44.68	80.07	96.3	99.15	99.61	99.9	99.96	99.98	99.98
Wine	56.46	90.41	99.32	99.9	99.9	100	100	100	100

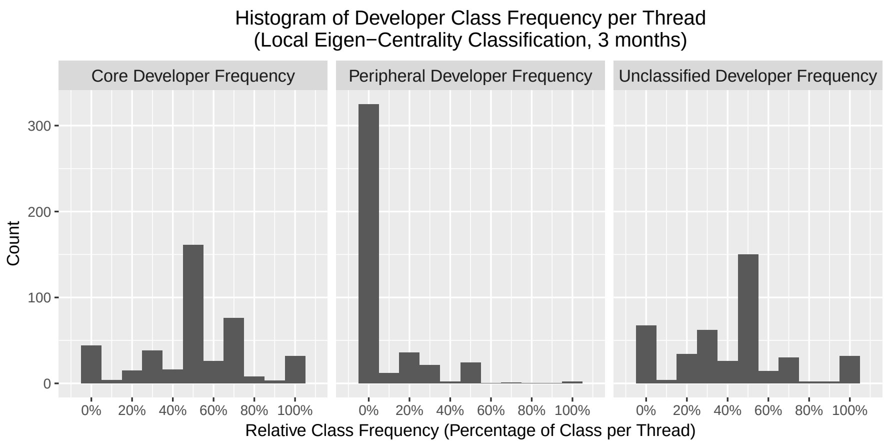

Fig. 4.2 Relative Thread Class Frequencies in Chromium

Our findings show that not only does $rThreads(P,3) > 50\%$ hold for all projects, but an overwhelming majority of threads – about 99% in all casestudies – contains 3 peripheral developers or less. In all projects except Chromium and U-Boot, a majority of threads is completely without peripheral contribution. This proportion – $rThreads(P,0)$ – ranges from 37% in Chromium to 89% in OpenSSL and we observe that in six of our subject projects, at least two thirds of the threaded mailing list discussions do not contain peripheral participants.

When investigating the occurence of unclassified members in mailing-list threads as shown in Table 4.5, we observe much more varied results for the different subject projects: In some, such as OpenSSL, the ML-only contributors play a very important part in the threads. For instance, we observe that, in OpenSSL, only 5 % of threads are without unclassified members and in a majority of threads, at least three different unclassified contributors participated. Nonetheless, there are some casestudies, in which the unclassified developers are a lot less active. One example is the project Ffmpeg where over 80 % of threads are completely without contribution by unclassified members and, even if ML-only contributors participate in threads, there is very seldomly more than 1 unclassified developer in a thread. There does not seem to be a general tendency regarding the level of unclassified involvement in threads in our 11 casestudies.

In Figure 4.2, we show the relative developer class frequencies in threads in the project Chromium, i.e., we indicate the absolute number of threads in which, e.g., 10 % of the participants are core developers. We observe that core members substantially shape most threads. However, peripheral members only constitute a very small proportion of the participants in most threads. There are almost no threads in which only one class of developers is present. It seems as though most threads in Chromium contain relatively equal proportions of core members and non-developers. In most projects, we observe a general phenomenon that core members significantly shape threads.

Tab. 4.5 Proportion of Threads with Maximum n Unclassified Developers: $rThreads(U, n)$ [%]

Casestudy	n								
	0	1	2	3	4	5	6	7	8
ApacheHTTP	33.93	75.74	92.87	97.24	98.63	99.35	99.72	99.79	99.88
Busybox	21.29	65.48	90.29	96.59	98.62	99.34	99.73	99.84	99.91
Chromium	26.49	75.88	93.18	97.27	99.07	99.57	99.78	99.93	99.93
Django	15.84	64.3	86.52	93.62	96.93	97.64	99.53	99.53	99.53
FFmpeg	82.86	98.06	99.84	99.94	100	100	100	100	100
Jailhouse	48.24	92.29	97.8	99.34	99.78	100	100	100	100
Llvm	18.06	65.59	89.87	96.71	98.47	99.33	99.82	99.88	99.88
OpenSSL	4.85	46.38	85.92	95.09	97.87	99.22	99.56	99.76	99.84
Qemu	61.25	86.07	95.59	98.15	99.04	99.46	99.64	99.79	99.88
U-Boot	76.48	96.22	99.29	99.85	99.96	99.98	100	100	100
Wine	81.51	97.46	98.83	99.71	99.8	99.8	100	100	100

Tab. 4.6 Variance of the Class Communication Longevity and Link-Intensity

Casestudy	$var(cLongevity(C_1, C_2))$				$var(cIntensity(C_1, C_2))$			
	(C,C)	(C,P)	(P,P)	(tot.,tot.)	(C,C)	(C,P)	(P,P)	(tot.,tot.)
ApacheHTTP	25 184	88 584	134 566	57 225	1.43	1.36	2.26	5.46
Busybox	28 464	46 277	30 530	30 217	3.71	9.11	9.64	13.2
Chromium	3 980	3 156	1 172	2 476	7.2	6.59	8.96	9.35
Django	2 875	2 918	1 035	2 014	0.99	11.87	3.55	5.7
FFmpeg	1 241	2 107	1 875	1 640	20.15	38.83	77.22	27.79
Jailhouse	788	1 909	4 609	3 518	759.25	252.71	778.35	271.24
Llvm	1 787	2 092	1 195	1 578	33.04	20.35	16.86	24.85
OpenSSL	147 278	137 153	20 581	11 608	1.17	2.24	4.11	9.41
Qemu	575	30 828	25 397	21 781	67.79	45.23	115.65	65.3
U-Boot	944	1 668	1 775	1 431	279.31	159.99	48.19	155.6
Wine	2 648	2 718	875	2 260	2.34	3.32	2.95	4.42

Given these points, we observe that, in most casestudies, peripheral involvement in mailing-list threads is very limited and we find that, in all of our casestudies, in a great majority of threads, no more than three different peripheral developers are involved. Hence, we **accept H1.3**.

4.4 Core-Peripheral Communication Episodes (H2.1)

We formulate our expectation that core-peripheral communication will be short lived but may be intensive in **H2.1**. We present our findings regarding the communicatino longevity in Table 4.9. Additionally, we show the communication longevity of only those communication episodes where multiple interactions occurred in Table 4.10. For the second part of our hypothesis, we show the relevant findings in Table 4.7 and Table 4.8. Table 4.7 contains the quantiles of the communication intensity based on the *links* created in the context of each communication episode, whereas Table 4.8 consists of the quantiles of the communication intensity based on the *mails* sent during communication episodes.

4.4.1 Communication Longevity

Regarding the communication longevity, we observe great differences in the core-peripheral communication episode longevity for different projects. The measured longevity ranges from a median of less than 1 day in Chromium to about 52 days, i.e., more than 7 weeks, in ApacheHttp. Although these two cases seem like extremes, there is no general tendency of, e.g., communication episode longevities lasting between 10 and 20 days.

In the first part of **H2.1a**, we anticipated that $q(cLongevity(C,P),0.5) < q(cLongevity(total,total),0.5)$ would hold. We observe that, in all subject projects except Ffmpeg and Wine, this is not the case. In these two projects, the median of the core-peripheral communication longevity is about 85% of the analogous value of the total-total project communication, i.e., $q(cLongevity(C,P),0.5) \approx 0.85 * q(cLongevity(total,total),0.5)$. In general, this *difference factor* varies between 1.2 in U-Boot and 58.29 in OpenSSL, although the latter as well as Django with a factor of 13.07 seem like outliers. In eight of our eleven subject projects, we observe a factor between 1.2 and 4, indicating that the typical core-peripheral communication episode is of greater longevity than a typical general communication episode by a factor of up to 4. Considering these values, we deduce that even for the two casestudies where our hypothesis is true, core-peripheral communication episodes are not significantly more short-lived than general communication episodes.

We observe similar results when analysing $cLongevityMulti(C,P)$, i.e., the communication longevity considering only communication episodes with at least two interactions, which we present in Table 4.10. In some subject systems, e.g., Chromium, this filtering has a significant influence on the median of the core-peripheral communication longevity, increasing it from 0.78 days considering all episodes to 14.44 days when the filtering is applied. Although the increase in communication longevity is considerable for all subject projects, in most, it is not as pronounced, increasing the value by about 10% to 25% in most projects. In JAILHOUSE the changes due to the applied filtering are the least significant, increasing the median of the core-peripheral communication longevity only by 2%.

Our findings indicate that this filtering acts as a normaliser for our data, as we now observe *difference factors* $f : q(cLongevityMulti(C, P), 0.5) \approx f \cdot q(cLongevityMulti(total, total), 0.5)$ between the different communication class combinations where $f \in [0.8; 4.1]$ instead of $f \in [0.8; 59]$ as observed for the unfiltered communication episodes. In all but two casestudies, the difference factor is now between 1 and 2, indicating that typical core-peripheral communication episodes are 1 to 2 times longer than typical total-total communication episodes when considering only communication episodes where "real" longevity could be measured, i.e., episodes which contain at least two interactions between which we could calculate the temporal distance to use as longevity.

In conclusion, we find that core-peripheral communication is *not* of shorter longevity than the general project communication including unclassified members. All things considered, we **reject H2.1a**.

4.4.2 Communication Intensity

When investigating the intensity of typical communication episodes between the core and the periphery, we observe values ranging from 0.07 links/day to 0.66 links/day. Most of our casestudies are characterised by a mean core-peripheral communication intensity between 0.1 and 0.4 links/day.

In **H2.1b**, we hypothesised that some core-peripheral communication episodes would be rather intensive. Hence, we anticipated that the inequality $q(cIntensity(C, P), 0.75) > q(c(Intensity(total, total), 0.75)$ would hold. As shown by the findings presented in Table 4.7, this assumption is generally not supported by our data. We observe only one subject project – FFMPEG – in which the core-peripheral communication reaches higher intensities than the general project communication. In this casestudy, the core-peripheral intensity is about 30 % higher than the general project communication intensity. Additionally, we find that the 0.75 quantile of core-peripheral communication-episode intensity equals the analogous value of total-total communication in CHROMIUM, LLVM, and U-BOOT. Nonetheless, in the majority of projects, the core-peripheral communication seems to be of rather low intensity compared to the general project communication intensity. In some cases, this difference is very pronounced. For instance, the core-peripheral communication is only about 20 % as intensive as the general communication in OPENSSL.

When we consider mails instead of links as indicator for the communication intensity, we get the results shown in Table 4.8. We observe that the switch from links to mails as intensity indicator decreases the intensity in all cases except CHROMIUM and WINE where it slightly increases the 0.75 quantile. In some subject projects, this decrease is quite drastic, such as Jailhouse where the 0.75 quantile of the core-peripheral mail intensity equals only about 22% of the analogous value for the link intensity.

The switch to mails as intensity indicator not only quantitatively influences the results, but also shows qualitative differences. Whereas the core-peripheral communication link-intensity in Ffmpeg was about 30% higher than the respective total-total communication link-intensity, we observe that, when using mails as intensity measure, the core-peripheral communication is only about half as intensive as the general project communication. Nonetheless, the findings using mail intensity support our hypothesis even less: Core-peripheral communication is more intensive than total-total communication in only one of the eleven subject projects, ApacheHTTP, although we see a very notable difference in this case as $q(cMailIntensity(C, P), 0.75) \approx 2.15 \cdot q(cMailIntensity(total, total), 0.75)$. However, in most casestudies, the 0.75 quantile of the core-peripheral mail-intensity is only about 30% to 50% of the comparative total-total value.

Overall, our findings indicate that core-peripheral communication is not more intensive than the general project communication – in most subject projects, the inverse effect is visible. This result is not changed by using mails instead of links as intensity measure. Although different casestudies support our hypothesis for both metrics, these findings only occur in a very small number of our subject systems. Given these points, we **reject H2.1b**.

4.5 Core-Core Communication Episodes (H2.2)

In **H2.2**, we hypothesised that communication between two core developers would be rather long-lived but varied in intensity. We present the findings concerning the communication longevity in Tables 4.9 and 4.10. Our results regarding the second part of the hypothesis are presented in Table 4.6.

4.5.1 Communication Longevity

In Table 4.9, we present our findings concerning the core-core communication longevity. We observe that, depending on the casestudy, the duration of typical core-core communication episode ranges from 19 days in Ffmpeg to 54 days in OpenSSL. In most subject projects, the median longevity of communication episodes between core developers is about 3 weeks to 4 weeks. Consequently, the differences in longevity between our eleven casestudies manifest significantly less strongly in the core-core communication than in the project-wide communication and especially the core-peripheral communication.

In **H2.2a**, we anticipated that a typical core-core communication episode would be rather long-lived according to the project's communication standards, i.e., that $q(cLongvity(C, C), 0.5) > q(cLongevity(total, total), 0.5)$

Tab. 4.7 Class Communication Link-Intensity, Quantiles

Casestudy	$q(cIntensity(D_1, D_2), x)$ [links/day]											
	(Core, Core)			(Core, Periph.)			(Periph., Periph.)			(Total,Total)		
	0.25	0.5	0.75	0.25	0.5	0.75	0.25	0.5	0.75	0.25	0.5	0.75
ApacheHTTP	0.05	0.14	0.33	0.02	0.07	0.22	0.01	0.06	0.4	0.03	0.12	0.52
Busybox	0.08	0.22	0.62	0.04	0.15	0.61	0.05	0.27	2	0.06	0.32	2
Chromium	0.06	0.15	0.54	0.06	0.26	2	0.28	2	3	0.09	0.56	2
Django	0.05	0.11	0.21	0.05	0.18	1	0.18	1.5	2	0.07	0.21	2
FFmpeg	0.17	0.46	1.42	0.12	0.56	2.59	0.33	1.58	4.8	0.15	0.55	2
Jailhouse	0.39	1.38	4.67	0.21	0.66	2.62	0.07	0.33	2.38	0.23	0.94	3.5
Llvm	0.11	0.28	0.9	0.1	0.38	2	0.14	0.8	3	0.14	0.65	2
OpenSSL	0.02	0.08	0.22	0.02	0.1	0.38	0.05	0.2	1	0.04	0.5	2
Qemu	0.14	0.4	1.33	0.07	0.26	1.2	0.1	0.47	2.12	0.1	0.38	2
U-Boot	0.22	0.6	2.21	0.15	0.44	2	0.13	0.45	2	0.17	0.53	2
Wine	0.05	0.12	0.38	0.04	0.18	1.33	0.08	0.3	1.38	0.05	0.2	1.5

Tab. 4.8 Class Communication Mail-Intensity, Quantiles

Casestudy	$q(cIntensity(D_1, D_2), x)$ [mails/day]											
	(Core, Core)			(Core, Periph.)			(Periph., Periph.)			(Total,Total)		
	0.25	0.5	0.75	0.25	0.5	0.75	0.25	0.5	0.75	0.25	0.5	0.75
ApacheHTTP	0.04	0.12	0.25	0.02	0.05	0.17	0.01	0.05	0.27	0.03	0.1	0.36
Busybox	0.06	0.17	0.35	0.03	0.12	0.36	0.03	0.2	2.4	0.05	0.23	1.65
Chromium	0.05	0.12	0.31	0.05	0.21	2.83	0.24	1.85	12.41	0.07	0.41	3.89
Django	0.04	0.1	0.19	0.04	0.15	0.9	0.15	0.87	3.93	0.06	0.18	1.72
FFmpeg	0.13	0.3	0.71	0.09	0.3	1.78	0.26	0.87	4.23	0.12	0.34	1.24
Jailhouse	0.26	0.51	1.1	0.14	0.3	0.57	0.06	0.12	0.94	0.15	0.4	1.61
Llvm	0.08	0.18	0.46	0.08	0.26	1.93	0.12	0.49	3.75	0.11	0.41	2.87
OpenSSL	0.02	0.06	0.2	0.02	0.08	0.28	0.04	0.14	0.74	0.04	0.35	3.9
Qemu	0.1	0.23	0.5	0.05	0.15	0.44	0.06	0.23	1.12	0.06	0.21	0.73
U-Boot	0.14	0.29	0.74	0.1	0.26	0.73	0.09	0.26	1.06	0.11	0.28	1.07
Wine	0.04	0.11	0.3	0.03	0.15	1.44	0.07	0.25	1.49	0.04	0.18	1.41

would hold. Our findings support this assumption in all 11 subject projects. In half the subject projects – namely ApacheHTTP, Ffmpeg, Jailhouse, Qemu and U-Boot, there is only a slight difference and the typical core-core communication episode lasts about 30 % to 80 % longer than a typical total-total communication episode. However, the other projects show considerably more pronounced differences, e.g., core-core communication exceeding the duration of project-wide communication by a factor of 100 in OpenSSL. This tendency is even more apparent regarding the comparatively shorter communication episodes, i.e., the 0.25 quantile: In this case core-core communication is of significantly greater longevity than total-total communication episodes in every single subject system. In addition, we observe that, in all projects but Qemu and ApacheHTTP, the 0.75 quantile of core-core communication longevity is greater than the analogous project-wide value.

When considering only the communication episodes with multiple interactions, we observe these same effects, albeit less pronounced. In this case, $q(cLongvityMulti(C,C), 0.5) > q(cLongevityMulti(total, total), 0, 5)$ holds in all casestudies except ApacheHTTP, where core-core communication lasts slightly shorter than general project communication, exhibiting a core-core communication longevity that is about 90 % of the project-wide communication longevity. In most casestudies, the core-core communication is more long-lived by a factor of 1.20 to 2. The most pronounced difference can be observed in OpenSSL, where the typical core-core communication episode lasts more than 5 times as long as its project-wide counterpart.

This tendency is even more apparent regarding the 0.25 quantile: In this case, core-core communication is of considerably greater longevity than total-total communication episodes in every single subject system, differing by factors of up to 21 times in OpenSSL. In most systems, the assumption even holds when considering the 0.75 quantile. Although we observe comparably small core-core longevities in ApacheHTTP, Busybox, Ffmpeg, and Qemu, in all other projects, we find $q(cLongvity(C,C), 0.75) > q(cLongevity(total, total), 0.75)$.

Our findings show that the difference in communication longevity between core-core and total-total communication episodes is very apparent in all 11 subject projects. When considering only communication episodes with multiple interactions or comparatively long-lived communication, i.e., the 0.75 quantile, the tendency is not as pronounced but still visible. We observe a particularly strong difference in longevity when comparing the 0.25 quantile, i.e., the comparatively shorter communication episodes. This holds when considering all communication as well as filtering out episodes with only one interaction. All things considered, we **accept H2.2a**.

4.5.2 Communication Intensity

We show our findings regarding the core-core communication intensity in Table 4.7. We observe that a typical communication episode between two core developers has an intensity ranging from 0.08 links/day in OpenSSL to 1.38 links/day in Jailhouse. Contrary to the communication longevity, these results show that the intensity difference between our casestudies manifests more strongly in the core-core communication than in the core-peripheral communication (see Section 4.4). This effect seems to be present regardless of which intensity metric is used. Nonetheless, there is no general tendency of how the intensity of a typical core-core communication episode, i.e., the median intensity, relates to the intensity of general communication. For link-based as well as for mail-based intensities, we observe some casestudies where core-core communication seems more intensive – e.g., Jailhouse as the project where this effect is most pronounced – as well as others where the inverse effect is observable, e.g., OpenSSL, in which the link-based and mail-based core-core communication intensity are only about 16 % as high as their total-total counterparts.

In addition, we observe that, in a small majority of projects, the 0.25 quartile of core-core communication link-based intensity is greater than the analogous value for the general project communication indicating that the less intensive core-core communication episodes exhibit higher intensities than the general project communication. This observation applies to all projects except Chromium, Django, Llvm, and OpenSSL where the core-core communication is slightly less intensive than the total-total communication as well as Wine where the two values do not differ. This effect is also present – albeit less pronounced – when investigating the mail-based communication. For this measure of intensity, we observe that – as for the link-based intensity – the 0.25 quantile of core-core communication mail-based intensity equals the analogous total-total intensity in Wine. In addition, the four projects in which link-based core-core intensity was less intensive than its project-wide counterpart – Chromium, Django, Llvm, and OpenSSL – show the same tendency when examining mail-based intensity. Thus, our observations concerning the core-core communication intensity do not change qualitatively when applying a different measure of intensity.

When examining the 0.75 quantile, we observe that the core-core communication is significantly less intensive than the total-total communication. This effect applies to all eleven casestudies when analysing mail-based intensity and to all casestudies except Jailhouse and U-Boot for the link-based intensity. Considering the link-based intensity, we observe five projects, where the 0.75 quantile of core-core communication episodes is less than half as intensive as the general project communication. The strongest difference can be observed in OpenSSL and Django, where the link-based core-core communication measures only 10 % of the general communication intensity. Moreover, we observe that the analogous values for the mail-based intensity indicate that the 0.75 quantile of the general project communication is between 1.5 and 20 times more intensive than the comparable core-core communication. Consequently, it seems as though core-core communication intensity shows comparably little variation.

This observation is also supported when analysing the variance of the core-core communication intensity shown in Table 4.6. We had hypothesised in **H2.2b** that $var(cIntensity(C, C)) < var(cIntensity(total, total)$ would hold. In fact, our findings show the inverse tendency: $var(cIntensity(C, C)) < var(cIntensity(total, total)$ holds in seven of our eleven subject projects. In some cases, this inequality is extremely pronounced, e.g., OpenSSL where the total-total variance is greater than the core-core variance by a factor of 8. Nonetheless, in Jailhouse, Llvm, Qemu, and U-Boot, our assumption that core-core communication intensity shows greater variation than its project-wide counterpart is correct. As explained above, these are also the projects, which show anomalies regarding the 0.25 and 0.75 quantile of the core-core communication intensity compared to their respective total-total communication intensity.

Interestingly, we do not see any correlation between the intensity-variance and the longevity-variance: Whereas there are some projects, such as OpenSSL, in which the core-core longevity-variance is considerably higher than its project-wide counterpart while the core-core intensity varies significantly less than the general project communication intensity, in other projects, e.g., ApacheHTTP, the inverse effect can be observed.

In summary, we observe that the intensity differences between subject systems manifest more strongly in the core-core communication than in the core-peripheral communication. In addition, we observe that the 0.25 quantile of core-core communication is slightly more intensive than the general project communication in most casestudies. When regarding the median, we do not observe any general tendencies. However, the 0.75 quantile of core-core communication episodes is considerably less intensive than its project-wide counterpart – in some cases they differ by a factor of up to 20. These tendencies are present regardless of the intensity metric used. Thus, we observe that – although it does not usually reach intensities as low as the general project communication – core-core communication also cannot attain the total-total communication's very high intensities. Generally, it seems as though core-core communication intensity is a lot more stable and balanced, with significantly less variation towards very low as well as very high values than the general project communication.

In general, our assumption does not hold, although there are a few projects in which we observe more varied core-core communication intensity than total-total communication intensity. As described above, these are the same projects, which show anomalies concerning the distribution of the core-core communication intensity compared to the project-wide values. It seems as though there are two types of projects regarding the core-core communication intensity. Nonetheless, we observed that the core-core communication intensity is rather stable and is characterised by little variation in a majority of our subject projects and, hence, we **reject H2.2b**.

4.6 Peripheral-Peripheral Communication Episodes (H2.3)

In **H2.2**, we formulated our expectation that communication episodes between two peripheral developers would be rather short-lived compared to the general project communication, i.e., we anticipated $q(cLongevity(P, P), 0.5) < q(cLongevity(total, total), 0.5)$ would hold. We show our findings in Table 4.9.

The longevity in a typical peripheral-peripheral communication episode ranges from 0 days in Chromium and Django to 20 days in Jailhouse. As described in Section 3.2, a communication episode is assigned a longevity of 0 days in case only one interaction occurred in the episode. Thus, these values could indicate that peripheral-peripheral communication often consists of only one interaction without any answers or follow-ups.

We observe that – except for Jailhouse and OpenSSL – our assumption holds. In most cases the difference in longevity is very pronounced and typical peripheral-peripheral communication episodes last less than half as long as their total-total counterparts. In ApacheHTTP, Qemu, and Wine, this difference is not as pronounced but still noticeable. Interestingly, we observe that peripheral-peripheral communication episodes in Jailhouse and OpenSSL are 1.4, respectively 25, times as long-lived as typical total-total communication episodes. This difference is especially pronounced in OpenSSL, which is not actually surprising considering that, in this project, core-core, respectively core-peripheral, communication lasted 90, respectively 58, times as long as the general communication episodes.

The tendency for peripheral-peripheral communication episodes to be of little longevity compared to project standards is even more apparent when regarding the 0.25 quantile, for which the peripheral-peripheral communication longevity is smaller than or equal to its total-total counterpart in every single casestudy. Interestingly, we do not observe similar tendencies when considering the 0.75 percentile, i.e., the comparably longer peripheral-peripheral communication episodes. Although the communication among peripheral project members is less long-lived in some projects, e.g., Django as the subject project where this phenomenon is the most significant, our findings for other projects, e.g., ApacheHTTP, indicate the opposite effect or, as in the case of U-Boot, no considerable difference in longevity can be found.

When considering only communication episodes with multiple interactions, we observe that our assumption holds in all casestudies except ApacheHTTP, Jailhouse, and OpenSSL for which peripheral-peripheral communication is about 60 %, 70 %, and 150 %, respectively, more long-lived than the general project communication. In all other casestudies, peripheral-peripheral communication is noticeably more short-lived. For the 0.25 quantile, the same effect can be observed in all subject projects except for the aforementioned three casestudies. Analogously to the longevity considering all communication episodes, no general tendency can be found regarding the 0.75 quantile of longevity in communication episodes with multiple interactions.

To summarise, we see that peripheral-peripheral communication are of comparatively short longevity in all but two casestudies. This effect is observable regarding the 0.25 as well as the 0.5 quantile. Our findings are not qualitatively changed by filtering out communication episodes with only one interaction, although it seems as though peripheral-peripheral communication is significantly marked by these short communication episodes. Overall, we **accept H2.2**.

4.7 Comparison of Core-Core, Core-Peripheral, and Peripheral-Peripheral Communication Episodes

When comparing the core-core, core-peripheral, and peripheral-peripheral communication episodes, we observe significant differences which pertain not only to the different levels of activity in the project core and periphery but also to the longevity and intensity of the communication. We present the corresponding results in the following.

4.7.1 Communication Longevity

When considering the communication longevity of all communication episodes presented in Table 4.9, we find that in all subject projects except ApacheHTTP and Qemu, a typical core-core communication episode is of greater longevity than core-peripheral communication episodes. In addition, we observe that core-core communication is more long-lived than peripheral-peripheral communication in all eleven casestudies. In fact, in nine of our eleven subject projects, $q(cLongevity(C,C), 0.5) > q(cLongevity(C,P), 0.5) > q(cLongevity(P,P), 0.5)$ holds, i.e., core-core communication is typically more long-lived than core-peripheral communication which is, in turn, of greater longevity than peripheral-peripheral communication. In most subject projects, these differences are very considerable, e.g., Django, in which the typical core-core, core-peripheral, and peripheral-peripheral longevity are 53 days, 34 days, and 15 days, respectively.

These tendencies are not only apparent in the typical communication episodes, i.e., the median of the communication-episode longevity, but are also visible in the distributions of communication longevities. We show these distributions for Ffmpeg and U-Boot in Figures 4.3 and 4.4, respectively. We only show the values in the 0.8 quantile, as our plots would otherwise be distorted and unreadable due to outliers. The distributions regarding the communication longevities seem very consistent across our subject projects, generally differing quantitatively but not qualitatively.

Tab. 4.9 Class Communication Longevity, Quartiles, All Communication Episodes

Casestudy	$q(cLongevity(D_1, D_2), x)$ [days]											
	(Core, Core)			(Core, Periph.)			(Periph., Periph.)			(Total,Total)		
	0.25	0.5	0.75	0.25	0.5	0.75	0.25	0.5	0.75	0.25	0.5	0.75
ApacheHTTP	16.23	31.72	72.97	15	51.99	169.05	0.03	18.91	150.71	0.51	21.21	85.45
BusyBox	16	29.82	67.2	9.1	28.8	98.64	0.01	5.29	47.85	0.01	8.01	43.08
Chromium	6.8	22.88	57.3	0	0.78	25.11	0	0	1.87	0	0.21	15.19
Django	14.52	27.82	71.41	0	7.45	38.63	0	0	0.93	0	0.57	21.05
FFmpeg	11.06	19.01	33.44	0.66	10.04	34.52	0	0.96	8.77	0.72	12.33	29.39
Jailhouse	13.4	24.91	36.71	13.52	21.29	37.12	0.34	19.92	43.04	1.19	13.92	29.08
Llvm	9.62	22.04	48.2	0.1	8.06	35.91	0	0.91	12.97	0	2.02	21.21
OpenSSL	17.09	53.55	172.62	7.34	34.39	138.12	0.06	14.84	57.86	0	0.59	24.88
Qemu	14.14	23.8	44.91	10.76	26.31	75.93	0.73	13.85	50.24	1.36	17.1	49.02
U-Boot	12.21	20.16	37.73	4.79	15.96	38.09	0.73	12.18	31.68	0.95	13.15	31
Wine	1.16	20.54	61.9	0	1.72	37.21	0	0.57	16.91	0	2.02	32.03

Tab. 4.10 Class Communication Longevity, Quantiles, Only Answered Communication Episodes

Casestudy	$q(cLongevity(D_1, D_2), x)$ [days]											
	(Core, Core)			(Core, Periph.)			(Periph., Periph.)			(Total,Total)		
	0.25	0.5	0.75	0.25	0.5	0.75	0.25	0.5	0.75	0.25	0.5	0.75
ApacheHTTP	16.87	32.31	74.21	20.8	61.63	184.06	11.88	55.54	234.18	11.42	34.89	113.6
BusyBox	16.19	29.98	67.82	12.63	34.77	110.68	1.63	16.35	96.63	2.79	18.94	70.71
Chromium	13	28.05	63.03	1.06	14.44	58.12	0.24	1.86	13.79	0.8	8.26	40.93
Django	16.53	29.88	76.66	6.25	16.39	59.23	0.55	3.43	15.94	1.83	14.98	50.93
FFmpeg	11.91	19.79	34.61	2.82	13.9	42.86	0.96	5.53	13.98	7.02	15.91	34.84
Jailhouse	13.4	24.91	36.71	13.79	21.8	37.71	7.5	26.97	62.03	6.44	15.35	32.98
Llvm	11.78	24.8	51.83	2.11	14.87	48.71	0.96	6.97	27.49	1.18	10.14	34.31
OpenSSL	17.92	56.24	182.19	12.91	42.45	165.55	6.73	26.53	70.68	0.84	10.46	89.08
Qemu	14.25	23.92	45.08	13.19	29.19	83.32	6.02	19.74	67.1	9.32	22.25	61.35
U-Boot	13.05	20.98	38.58	8.52	18.86	41.9	5.58	15.27	37.03	7.18	16.87	37.1
Wine	10.99	26.24	69.95	2.59	20.74	65.59	1.86	13.36	40.73	2.18	18.9	62.05

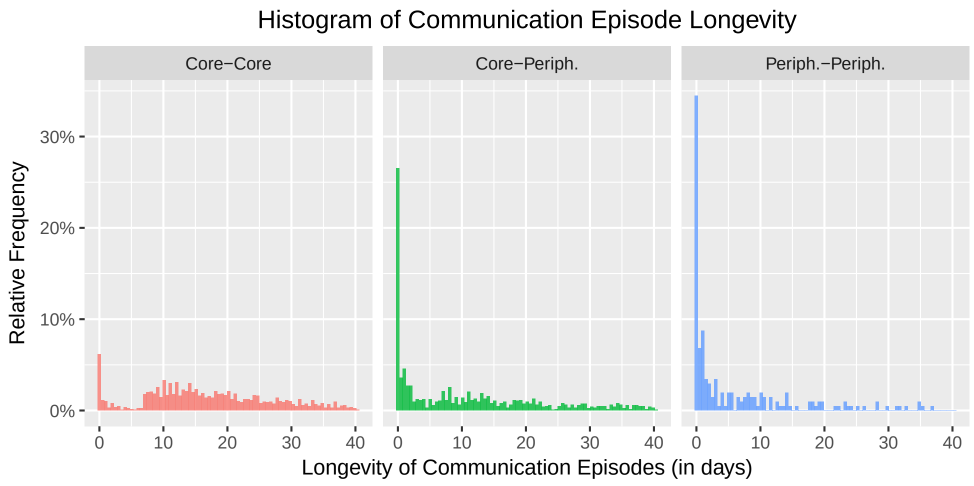

Fig. 4.3 Class Communication Longevity in FFMPEG, not showing the highest 20% of values

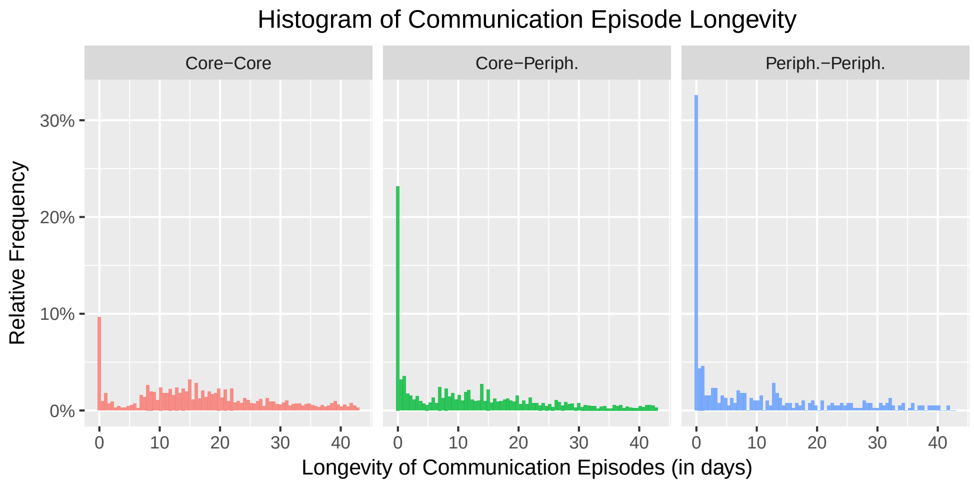

Fig. 4.4 Class Communication Longevity in U-BOOT, not showing the highest 20% of values

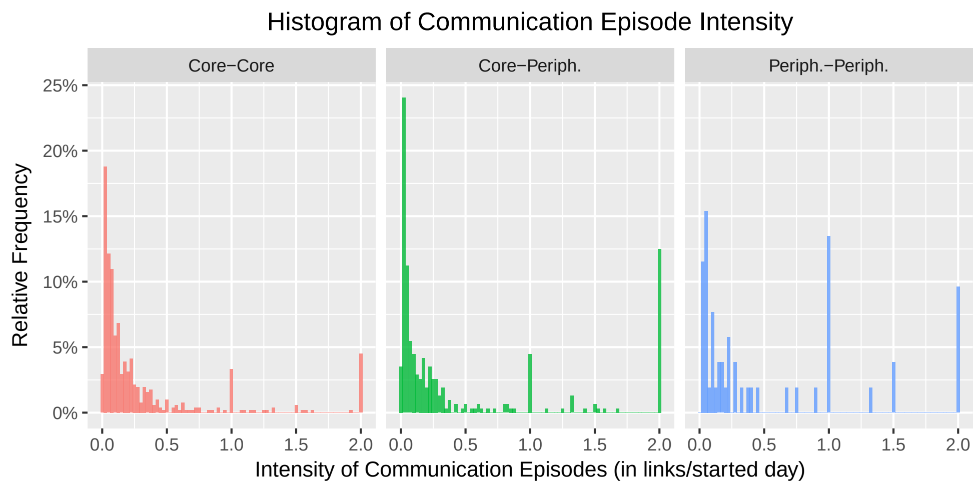

Fig. 4.5 Class Communication Link-Intensity in Wine, not showing the highest 10% of values

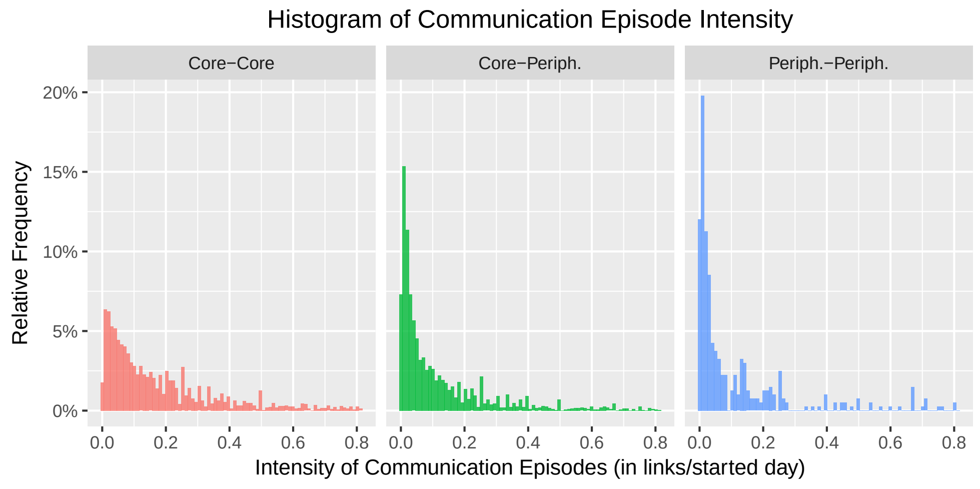

Fig. 4.6 Class Communication Link-Intensity in ApacheHTTP, not showing the highest 10% of values

4.7.2 *Communication Intensity*

Considering the link-intensity of the typical communication episodes as presented in Table 4.7, we do not observe any tendencies that are as clear. Nonetheless, we find that peripheral-peripheral communication is typically the most intensive form of communication in all projects except ApacheHTTP, Jailhouse, and U-Boot. In these three projects, core-core communication link-intensity surpasses core-peripheral as well as peripheral-peripheral communication intensity, in the case of Jailhouse significantly so. A similar tendency is visible when investigating the mail-intensity presented in Table 4.8: In eight cases, the peripheral-peripheral communication episodes typically show the greatest intensity. In the remaining three projects, core-core communication surpasses all other communication in intensity. These are the same projects in which the core-core communication dominates the other communication types with regards to the link-intensity. Thus, we observe that the switch to a mail-based activity metric only quantitatively changes our findings but the tendencies we observe remain stable. Interestingly, the switch of intensity metric intensifies differences in some cases and smoothes them in others.

As there are no qualitative differences between the link-based and the mail-based communication intensity, we only discuss the link-based results in the following. In Figures 4.5 and 4.6, we show the results for one project in which peripheral-peripheral communication is the most intensive, Wine, as well as one project in which core-core communication dominates, ApacheHTTP, respectively. We show only the 0.9 quantile in the figure as high outliers would otherwise distort the plots. The tendencies we observe for typical intensity, i.e., the median intensity, are reflected in the intensity distributions. In Wine, we clearly see that a great majority of core-core communication is of comparatively little intensity. This effect is visible and even intensified in the core-peripheral communication episodes. In ApacheHTTP, however, we see the exact opposite tendency: peripheral-peripheral communication is typically the least intensive, followed by the core-peripheral communication intensities. We observe clearly that core-core communication generally constitutes the most intensive communication episodes.

To summarise, we find that core-core communication episodes are generally of greater longevity than core-peripheral communication which is, in turn, more long-lived than peripheral-peripheral communication. For the most part, we observe that the inverse is true when considering communication intensity: In eight subject projects, peripheral-peripheral communication episodes are the most intensive. In the remaining three, we observe that core-core communication is of the greatest intensity. These findings do not change qualitatively when using a mail-based intensity metric. Interestingly, the results we observe for the typical communication episodes are also reflected in the distributions of the communication longevity and intensity.

4.8 Average Class Activity (H3.1)

In **H3.1**, we hypothesised that code contributors would be more active regarding the number of mails written than mailing-list-only contributors. Thus, we expected the inequalities $aMails(C) > aMails(U)$ and $aMails(P) > aMails(U)$ to hold. We present our findings concerning the average number of mails written by members of the three classes and additionally the average number of threads started in Table 4.11.

We observe that the two inequalities hold in all eleven subject projects. In fact, our findings even show that there is a strict order of activity in which the average core developer is more active than the average peripheral developer who, in turn, is more active than the average non-developer, i.e., $aMails(C) > aMails(P) > aMails(U)$. Except for Chromium, this hierarchy of activity is also visible in the average number of threads started by members of the three classes.

As we not only observed that core and peripheral members are more active than unclassified members regarding the average number of mails written, but even found evidence for a strict hierarchy of activity in all eleven subject projects considering e-mail activity and in all but one project considering threads as an activity indicator, we **accept H3.1**.

4.9 Activity Level of the Core Class (H3.2)

In **H3.2**, we theorised that core developers were responsible for the bulk of the communication activity. We anticipated that this would pertain to the proportion of mails written by core members as well as the proportion of threads started (out of all threads that received an answer). The proportions of mails written and threads started for all subject systems and all three participant classes are shown in Table 4.12.

Regarding the proportion of mails written as indicator for the class activity, we find that core members are responsible for the majority of mails in 6 of our 11 casestudies. In 3 of the subject projects – Busybox, Django, and Llvm – core members amounted for 40 % to 50 %. In Chromium and OpenSSL, core members were only responsible for 37 % and 22 %, respectively. Although our assumption is true for a slight majority of subject projects, we do not really see a general tendency towards core developers contributing the majority of e-mails.

Although core developers are generally not responsible for more than half the mails, we observe that they are usually more active than the other two developer groups: In all projects except OpenSSL, we find that the proportion of mails contributed by core developers exceeds the comparable values for the peripheral and unclassified member groups. When comparing the periphery and the ML-only contributors, we do not see any clear tendencies. In 7 casestudies we observe $rMails(U) > rMail(P)$ but in the remaining 4, the opposite effect is visible. Hence, we do not observe any tendency that affects

Tab. 4.11 Average Number of Mails and Average Number of Threads Started per Class Member

Casestudy	*aMails*(*D*)				*aThreadsStarted*(*D*)			
	Core	Periph.	Unclass.	Total	Core	Periph.	Unclass.	Total
ApacheHTTP	32.09	8.93	4.98	9.99	8.92	2.9	2.32	3.51
Busybox	35.31	7.09	4.41	7.92	4.83	2.38	1.7	2.09
Chromium	16.51	3.73	2.99	4.7	2.01	0.75	1.03	1.03
Django	19.18	4.94	2.64	4.9	6.84	2.56	1.09	1.94
FFmpeg	140.64	11.42	5.02	27.87	25.68	2.61	1.4	5.52
Jailhouse	124.13	19.53	11.07	31.57	11.32	2.41	4.88	5.76
Llvm	24.09	6.45	4.34	7.21	3.93	1.25	1.19	1.55
OpenSSL	23.66	10.68	4.82	6.04	9.86	4.29	3.02	3.43
Qemu	219.69	31	6.66	26.7	24.07	5.07	2.14	4.29
U-Boot	219.27	26.93	6.45	33.66	40.6	6.63	1.99	7.28
Wine	35.82	6.43	3.44	12.75	19.82	2.4	1.1	6.33

Tab. 4.12 Relative Developer Class Activity

Casestudy	Mails [%]			Threads started [%]		
	Core	Periph.	Unclass.	Core	Periph.	Unclass.
ApacheHTTP	55.51	7.30	37.19	43.89	6.74	49.37
Busybox	48.08	5.82	46.10	24.97	7.42	67.61
Chromium	37.55	28.04	34.41	20.81	25.53	53.66
Django	48.30	9.55	42.16	43.45	12.47	44.07
FFmpeg	75.70	16.03	8.27	69.82	18.52	11.66
Jailhouse	68.48	5.91	25.61	35.21	4.00	61.79
Llvm	42.47	15.27	42.17	32.24	13.90	53.85
OpenSSL	21.95	4.71	73.34	16.08	3.32	80.60
Qemu	57.30	24.83	17.87	39.06	25.25	35.69
U-Boot	59.49	30.38	10.13	50.93	34.60	14.48
Wine	71.34	18.35	10.30	79.53	13.81	6.66

all projects regarding the relation between the contribution by the periphery and by unclassified project members.

When considering the number of threads started as activity metric, our findings indicate that core developers are usually not responsible for the bulk of communication activity. In fact, core members initiate the majority of mailing-list threads in only three of our casestudies, FFMPEG, U-BOOT, and WINE. Interestingly, there does not seem to be a direct correlation between the proportion of mails written and of threads started: Although we observe $rMails(C) < rThreads(C)$ in all casestudies except WINE, the proportion of mails written is no indicator for the proportion of threads initiated by core members. This becomes very apparent when comparing BUSYBOX and DJANGO which both exhibit 48 % as proportion of mails contributed by core members. Nonetheless, core members start 24 %, respectively 43 %, of threads in the two projects, differing by a factor of nearly 2.

Regarding hierarchy of activity as measured in threads started, we observe very different tendencies than considering e-mails. In seven of our casestudies, unclassified project members initiate more threads than the project core and periphery. In five systems, this group even starts more than half the mailing-list threads. However, we also observe three projects in which ML-only contributors are the least active group. Thus, we do not observe any general tendencies, but it becomes apparent that core members are generally not the most active group with regards to the initiation of mailing-list threads. Nonetheless, we observe that core members are responsible for more contributions than peripheral members in all subject projects except CHROMIUM. Thus, we see that the order between core and periphery established based on the source code is reflected in the mailing-list activity.

In essence, we observe that the project core is the most active group considering e-mail contributions, albeit not as active as we had anticipated. When considering the proportion of threads initiated, we do not observe these tendencies. Interestingly, there does not seem to be a direct correlation between the proportion of mails and of threads. In a majority of our subject projects, the unclassified members were the most active group when considering threads started. Additionally, we observe that the core-peripheral classification, i.e., the difference in activity established based on the source code, is reflected in the mailing-list activity, as the core group is generally more active than the periphery. All points considered, we **reject H3.2**.

4.10 Core Communication Activity during the Release Cycle (H3.3)

In **H3.3**, we hypothesise that the number of mails written by core developers will increase significantly prior to a release. We show the relative class communication activity for four subject projects in Figures 4.7, 4.8, 4.9, 4.9, 4.10, 4.11, and 4.12. The vertical, dotted lines in the plots indicate major releases.

We do not observe any release-specific patterns concerning the proportion of mails written by the project member classes in Ffmpeg, as shown in Figure 4.9. However, the evolution of the class activity supports our findings regarding the classes' proportions of contribution activity: The aggregated results we described in Section 4.9 are strongly reflected in the more fine-granular evolution of relative activity. We observe that core participation remains at an extremely stable 65 % to 85 %, exceeding this range only very seldomly. The same effect can be observed for the periphery's and the unclassified members' activity proportions of 10 % to 25 % and 5 % to 20 %, respectively.

We observe this same phenomenon in Chromium, as shown in Figure 4.11. Although the actual proportions of class activity differ considerably from those in Ffmpeg, we observe the same effect of stability. In Chromium, there is a nearly equal distribution of mail contributions among the three classes: the project core is generally responsible for 30 % to 50 % of the mailing-list activity, whereas the periphery and the unclassified members amount for 20 % to 35 % and 25 % to 40 %, respectively. Thus, we observe a division of contributions among the three group that is project-specific but very constant in these two projects.

This observation is supported by our findings regarding other subject projects: Despite the fluctuations, the relative core mail-activity generally does not exceed a range of size 20 %. We observe this stability in the proportion of contributions even for the unclassified and peripheral members in almost all subject projects. Admittedly, there are exceptions: In Qemu as well as Busybox, we observe several different phases of the class activity. We show the relative class activity in Qemu and Busybox in Figure 4.7 and 4.8, respectively.

In Qemu, there are two very distinct phases of the developer class activity: Up to release "release_0_10_0", i.e., the beginning of 2009, the unclassified project members contribute the bulk of the activity, the project periphery is nearly inactive and the core developers only amount for 0 % to 40 % of the messages posted to the development mailing list. However, this changes drastically. After abovementioned release, the core developers consistently contribute 50 % to 70 %. In addition, the periphery gains in importance, amounting for 20 % to 40 % of the mailing list traffic. The unclassified members become the least active group on the mailing list.

Thus, although the project exhibits stronger fluctuations in the beginning of the analysed ML-history and its class activity proportions change completely at one point in time, we observe the same role stability as in the other casestudies afterwards. Hence, our results in Qemu introduce the notion of different phases in the class activity, but otherwise support our conclusions regarding the stability of class activity proportions. Busybox, however, is the only subject project, in which this tendency is not directly observable. Although it seems as though there are different phases within which the class activity proportions remain comparatively stable, this stability is no longer visible in the data after release "1_18_0". Nonetheless, we do observe stable activity proportions divided in phases of at least 16 months each. The changing points of these phases usually seem to coincide with the major releases, namely "1_01", "1_2_0", "1_7_0", and "1_15_0". Thus, we still observe stability – albeit in changing, but relatively long-lasting phases – in most of the Busybox history.

Generally, we observe a relative high stability in the proportion of contributions. However, when considering the absolute number of mails written in Ffmpeg – presented in Figure 4.10 – we observe considerable fluctuations. This effect seems to affect primarily the project core, although peripheral developers show similar, albeit considerably less strong, fluctuations in the same time periods. The unclassified developer group is not affected by this tendency. Despite the occurence of these fluctuations, they do not seem to support our hypothesis: In most release cycles, e.g., after release "n2.8" and "n2.6", we observe a decrease in core activity and then – about 4 weeks into the release cycle – considerable spikes in the core activity. Regarding the beginning and end phase of the release cycles, we do not observe any effects that are consistent over time.

The absolute numbers of mails written per class for Chromium, which we present in Figure 4.12, show some similar tendencies. We also observe that fluctuations in the core activity are usually reflected in the other classes' activity level. In Chromium, this phenomenon seems to affect the unclassified members more strongly than the project periphery. In most cases, we observe uprises in core activity circa 2 weeks to 3 weeks after major releases. Generally, the participant activity – particularly the core activity – seems to go down towards the end of a release cycle. Nonetheless, no release-specific core activity patterns are visible in the other subject projects.

To summarise, our most interesting observation concerning the classes' activity is the stability in the proportion of their contributions. These proportions seem to be project-specific but very stable for most of our subject projects. Very long project histories might exhibit different, relatively long-lasting phases of this activity stability. In addition, we see two projects – Qemu and Busybox – that gain, respectively lose, this stability at a specific point in the project evolution. In some projects, we observe a recurring rise in general and especially core activity about 2 weeks to 4 weeks after major releases, but in most casestudies, no consistent tendency can be observed. In addition, we observe a decline in the core group's mail-activity towards the end of a release cycle, i.e., prior to a release, in several cases. Hence, we **reject H3.3**.

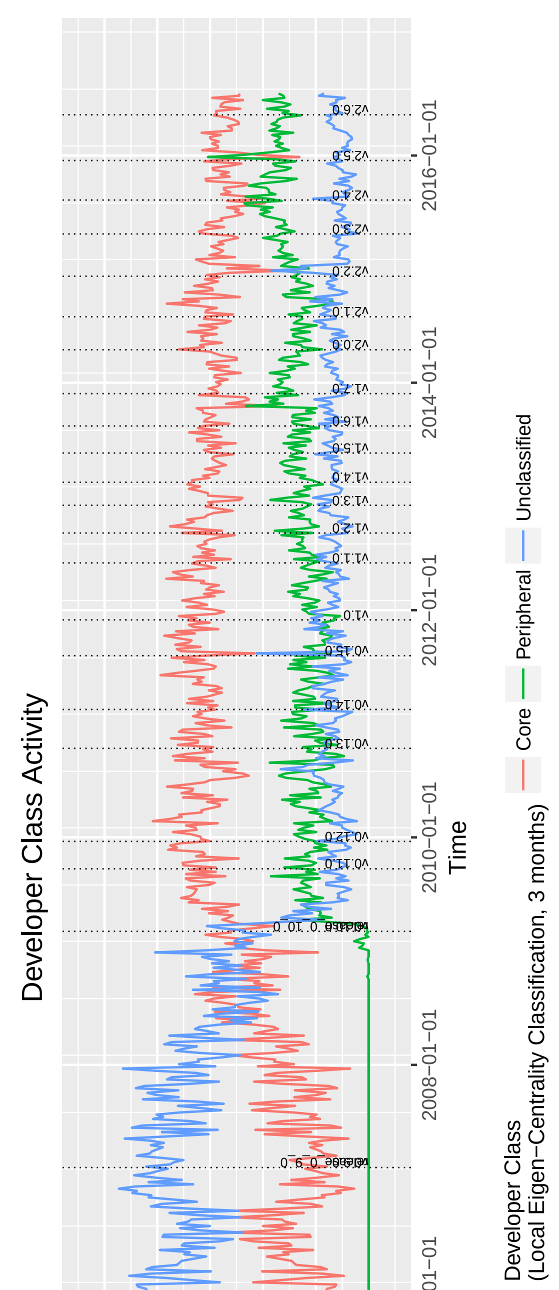

Fig. 4.7 Relative Class Activity in Qemu

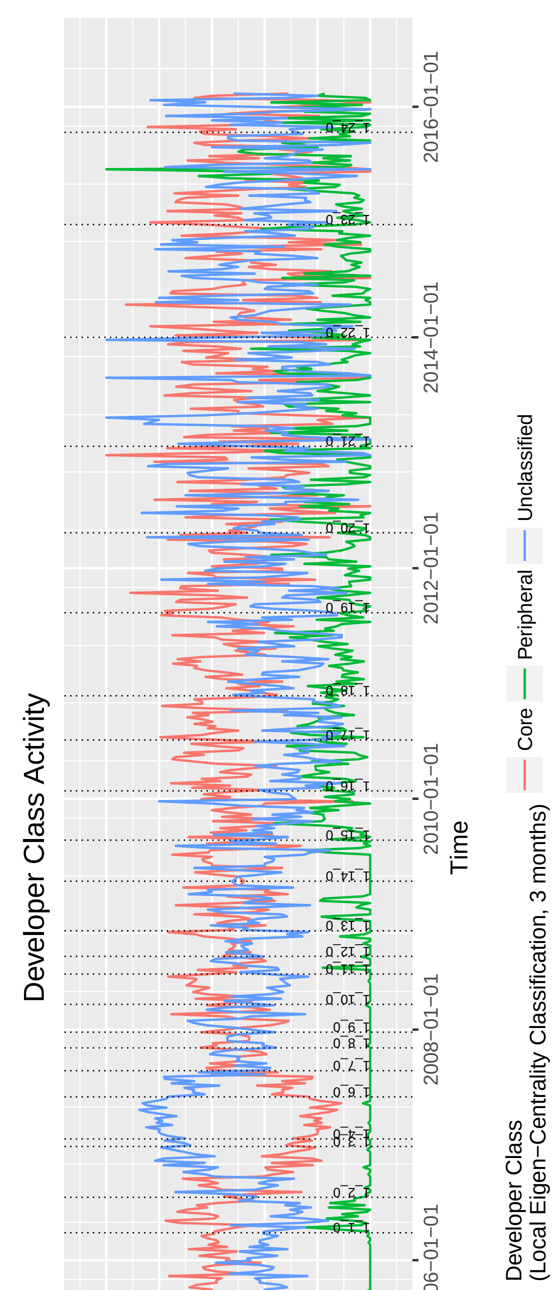

Fig. 4.8 Relative Class Activity in Busybox

4.11 Core Communication Activity during the Release Cycle (H3.3)

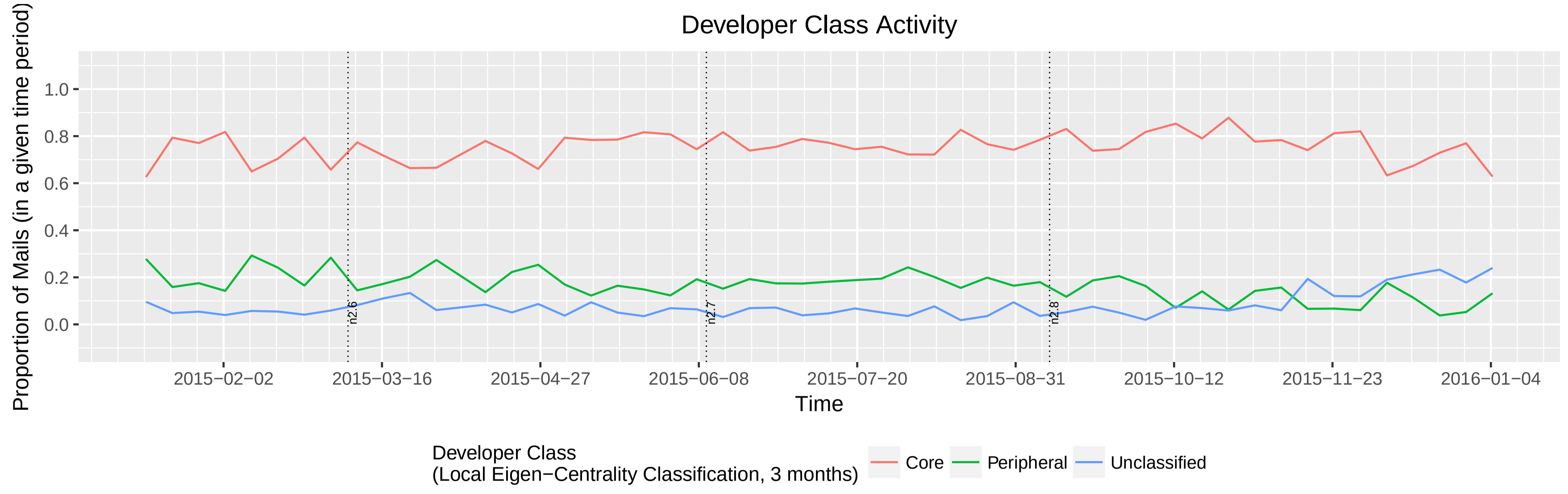

Fig. 4.9 Relative Class Activity in Ffmpeg

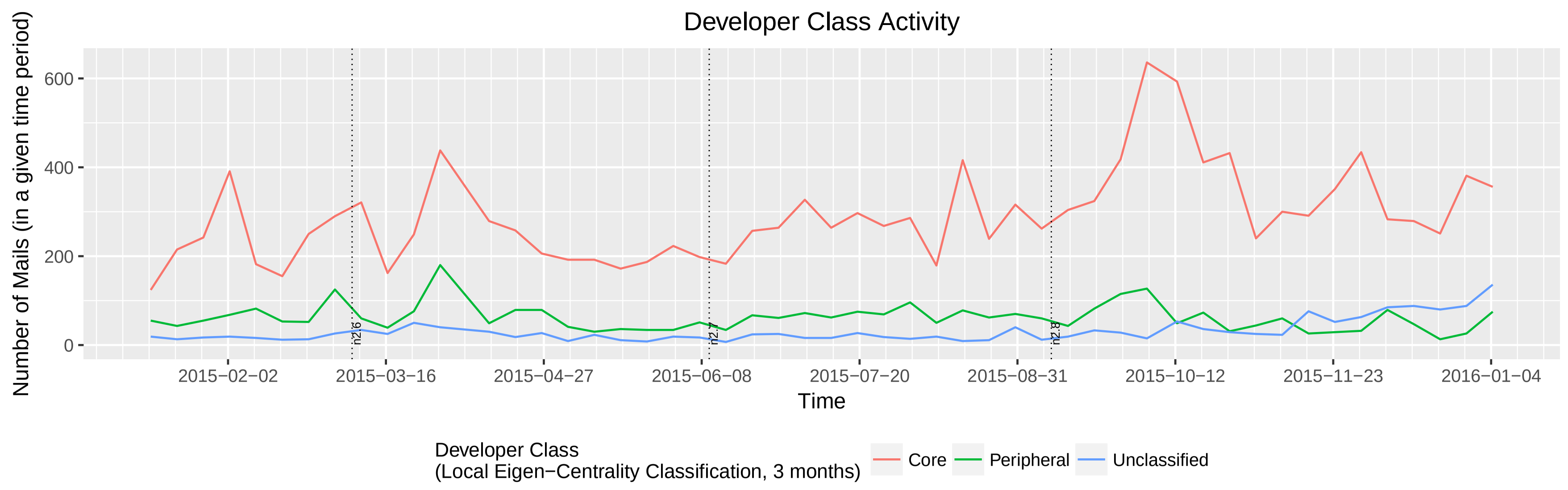

Fig. 4.10 Absolute Class Activity in Ffmpeg

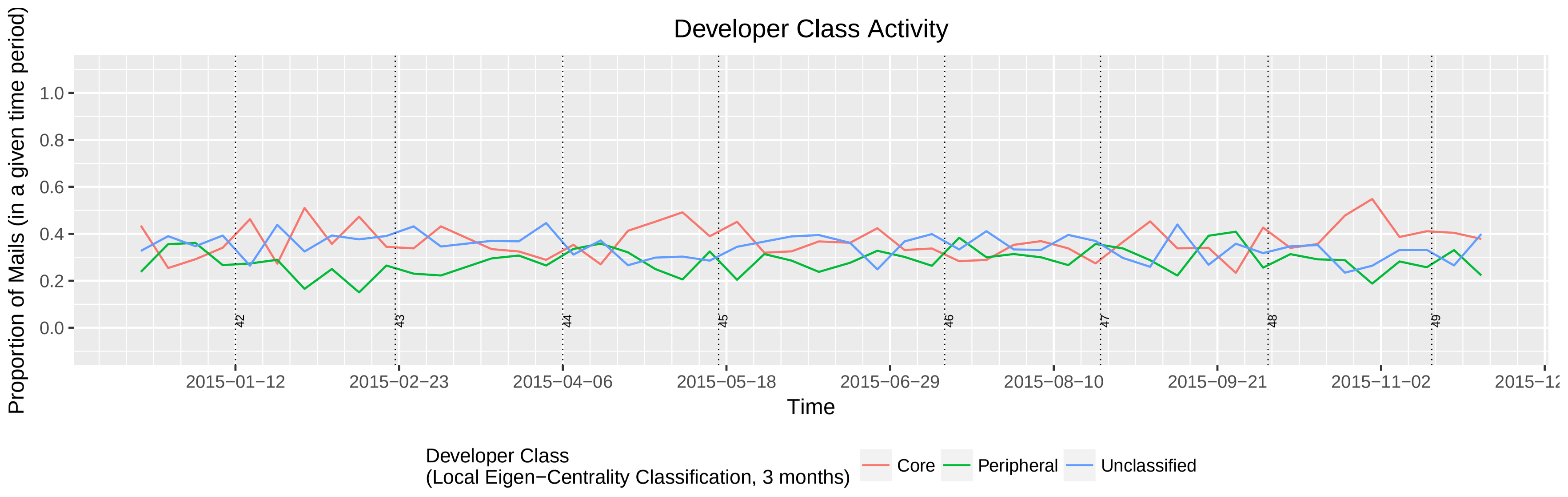

Fig. 4.11 Relative Class Activity in Chromium

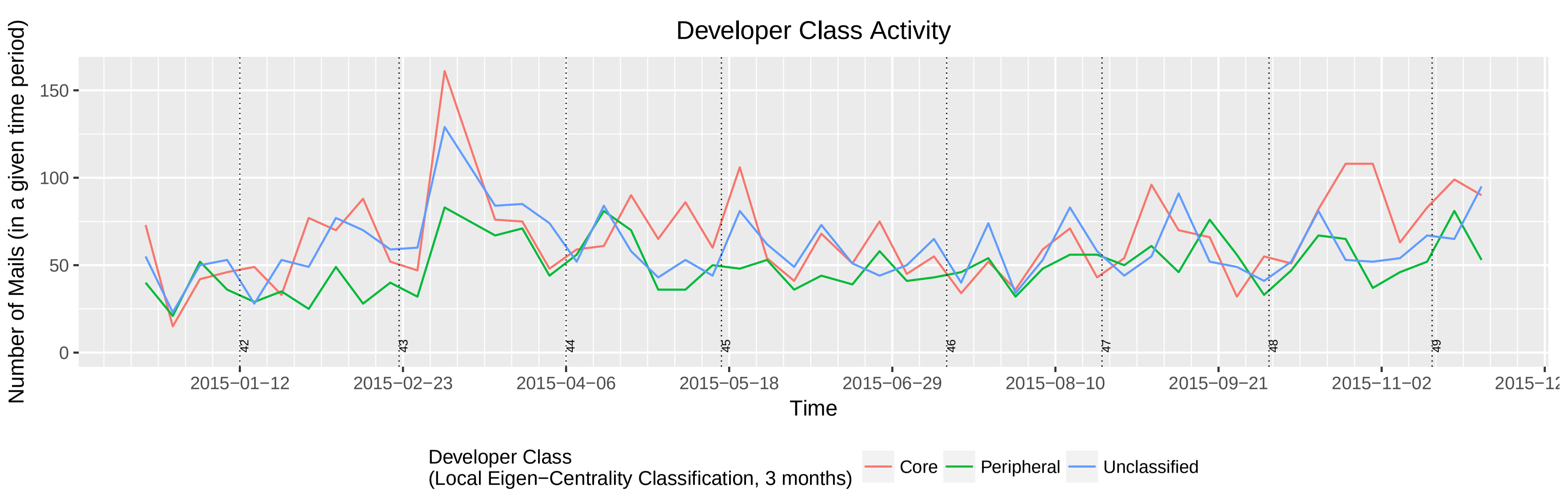

Fig. 4.12 Absolute Class Activity in Chromium

4.11 Peripheral and Unclassified Communication Activity during the Release Cycle (H3.4)

In **H**3.3, we formulated our hypothesis that the number of mails written by peripheral developers would spike during the beginning and end phase of the release cycle. We show the relative and absolute class communication activity in Busybox, Chromium, Ffmpeg, and Qemu in Figures 4.7, 4.8, 4.9, 4.9, 4.10, 4.11, and 4.12.

As explained in Section 4.10, we generally do not observe significant changes in the relative class activity, i.e., the proportion of mails each of the classes contributes to the project communication. This not only affects core members but also the project periphery and non-developers.

Considering the absolute class activity, i.e., the number of mails written by developers from each of the three groups, we do not observe any clear release-specific effects in the core and unclassified activity either. In Ffmpeg, there are only few noticeable increases in unclassified and peripheral activity as shown in Figure 4.10. These do not correspond to specific points in the release cycle. Although we observe spikes in peripheral mail activity directly prior to release "n2.6" as well as about 3 weeks to 4 weeks after releases "n2.6" and "n2.8", these seem coincidental and don't seem to to be linked to specific points in the release cycle.

Investigating the absolute class activity in Chromium, which we present in Figure 4.12, we do, however, see some tendencies that might indicate effects that the release cycle has on the peripheral activity: Although there are no general upturns in the beginning and end phase of release cycles, the data exhibits slight increases about 2 weeks after major releases. In addition, we observe an uprise in the last week prior to a major release in most cases. The increase in activity which takes place 2 weeks into the release cycle seems to be a phenomenon that affects all contributor groups in Chromium, whereas only the project periphery exhibits the slight upturns directly prior to most major releases.

In most projects, the relative class activity remains stable and is unaffected by the release cycle. In addition, our data does not indicate the existence of general release-specific tendencies. Although, we observe two potentially release-specific effects in Chromium, which affect all developer groups and primarily the periphery, respectively, our data is not sufficient to substantiate this assumption as the phenomenon could just consist of random fluctuations. In addition, these are not general changes in activity in certain phases of the release cycle but only two specific points in time: the week prior to a release as well as the point about 2 weeks into a release cycle. All things considered, we **reject H**3.4

4.12 Class Interaction during the Release Cycle (H3.5)

In **H3.5**, we hypothesised that core-core communication activity would increase in the time period prior to a release. In Figures 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, and 4.19, we show the class interaction, i.e., the edge class frequencies, in the context of release cycles for 5 notable subject projects. The vertical dotted lines in the plots indicate major releases.

As explained in Section 4.1, we observe that core members are consistently preferred by all project member groups in Ffmpeg. We present the relative edge class frequencies in Ffmpeg in Figure 4.1. Thus, we find that 60 % to 90 % of core communication is directed at other core members. Due to this consistently high number, there are not many fluctuations to be seen – especially not any patterns with regards to the release cycle. It seems as though core-core communication increases prior to release "n2.7" starting about 6 weeks before the point of release, but this tendency is not reflected in any of the other release cycles we investigate. Interestingly, the proportion of links sent to the core group originating from the peripheral as well as the unclassified participant group also increase in the same time span, but even more strongly. Therefore, this increase does not seem to present a pattern of core-core communication but of the general project communication and the importance or activity of the core group at that point in time.

Considering the project U-Boot, we see slightly different tendencies, as shown in Figure 4.13. Although core members are significantly preferred as communication partners by other core members, this tendency is not as pronounced and consistent in the periphery and the non-developers. In addition, we see significantly more fluctuations in the relative core-core interaction than was the case in Ffmpeg. Nonetheless, we do not observe any increases in the relative core-core communication prior to releases – in some cases, e.g., release "v2015.04" and "v2015.07", the inverse tendency seems to come into effect.

This observation is supported by the absolute class interaction we present in Figure 4.14. Although there are some increases in the core-core communication, these mostly take place in the first part of the release cycle. Even in the cases where we see small upticks directly prior to releases, the core-core activity is usually lower than in the first half of the release cycle.

In Figure 4.15, we show the analogous statistics for the project Wine. The findings are difficult to interpret as we observe only two major revisions which are spaced about 1 week apart due to Wine's release system. Thus, it might be a possibility that our results are slightly distorted by these two releases. When investigating the relative edge class frequencies, we do not find any noticeable changes in the core-core communication prior to the releases. However, considering the absolute developer class interactions shown in Figure 4.16, we observe a very noticeable spike in the absolute number of core-core communication about 1 month before release "wine-1.8". Although this increase is accompanied by a general rise in core activity – apparent in the simultaneous rise of core-core, core-peripheral, and core-

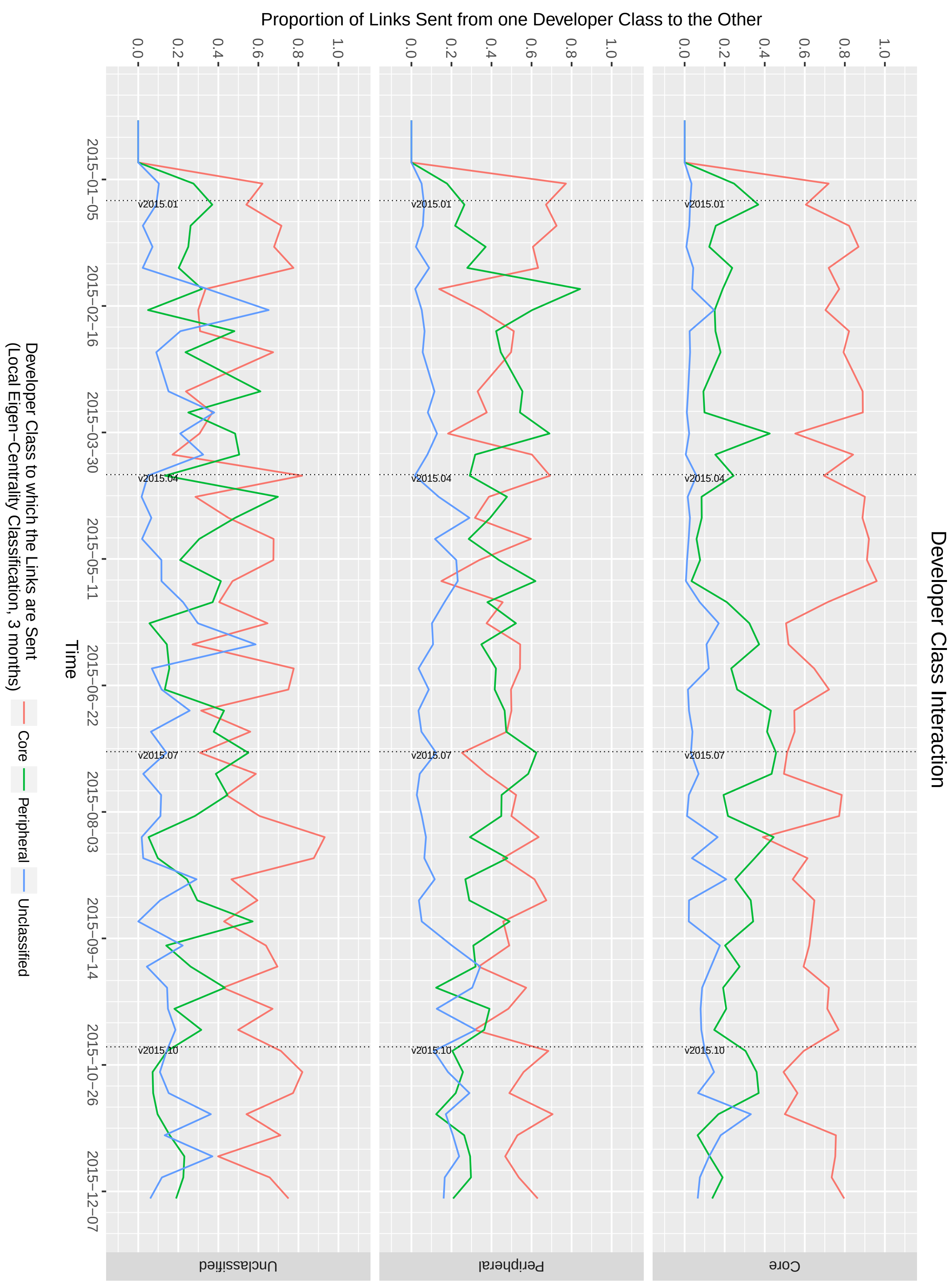

Fig. 4.13 Relative Edge Class Frequency Evolution in U-Boot

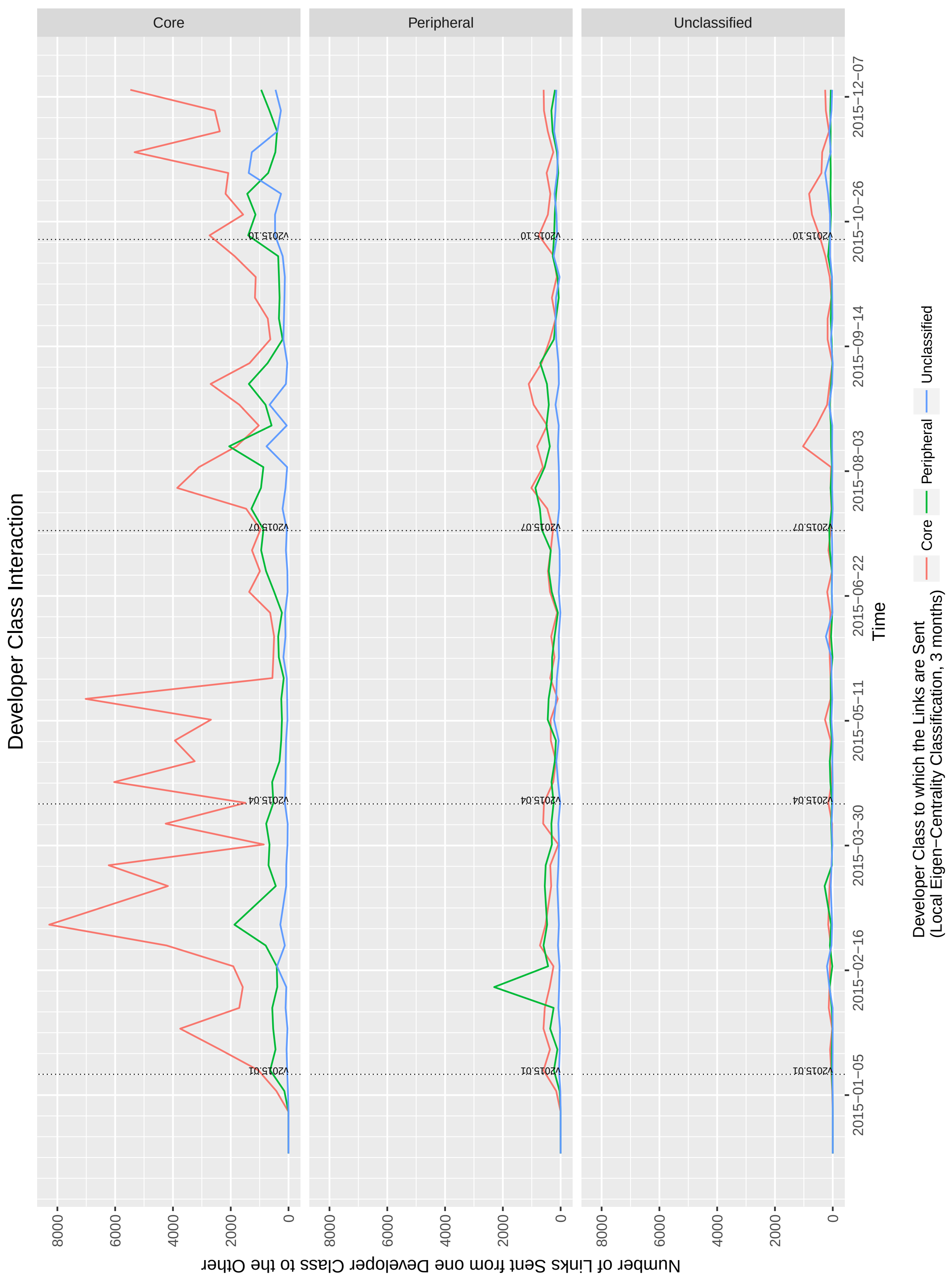

Fig. 4.14 Absolute Edge Class Frequency Evolution in U-Boot

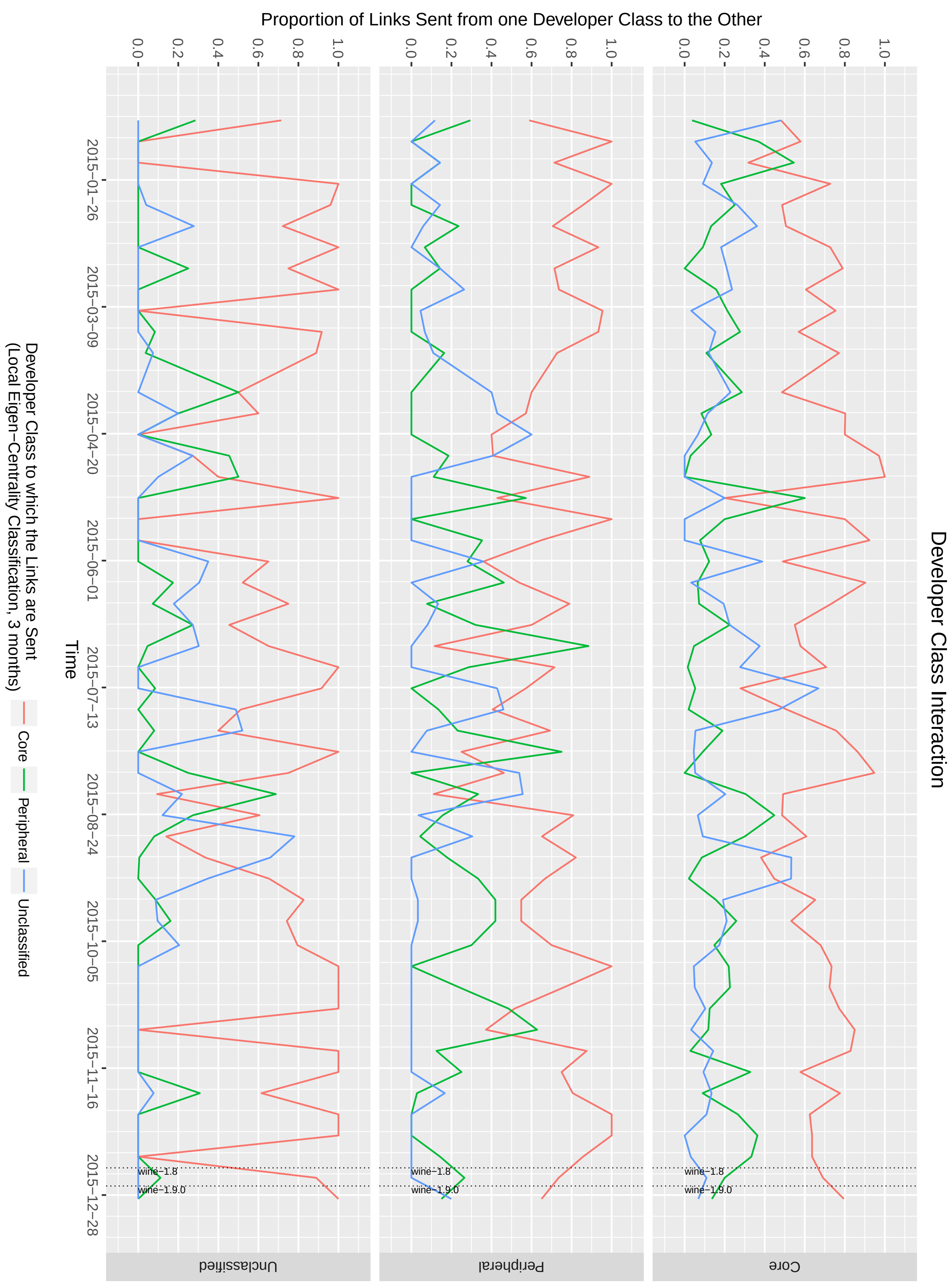

Fig. 4.15 Relative Edge Class Frequency Evolution in WINE

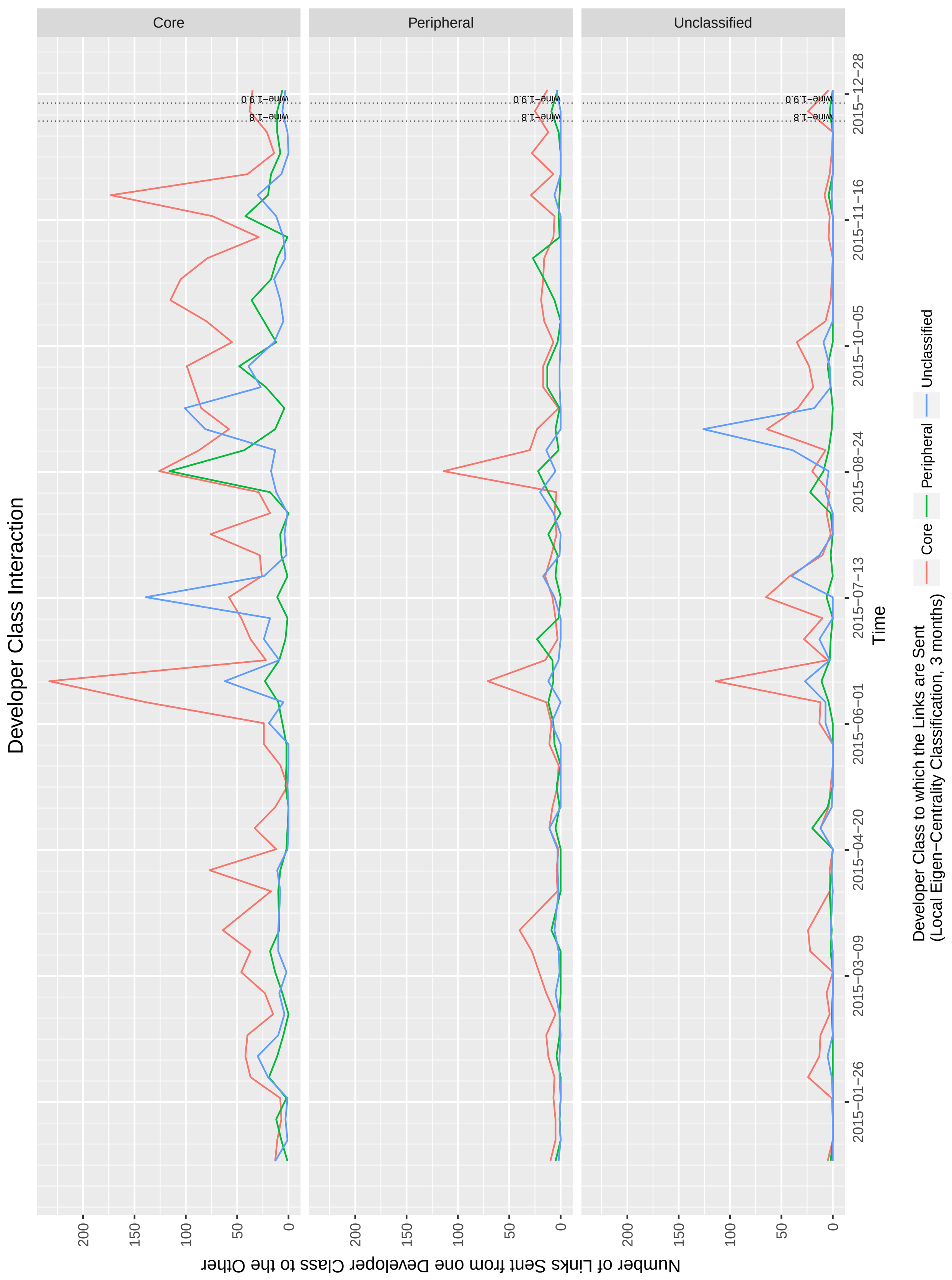

Fig. 4.16 Absolute Edge Class Frequency Evolution in Wine

unclassified communication – it affects the core-core communication most significantly. Although this is only one occurence and we do not have enough data to draw conclusions for general tendencies in Wine, this development could support our hypothesis.

Interestingly, we observe several spikes in the absolute class interaction in Wine. These upturns seem to originate from all developer classes, but apparently increase mostly the number of links sent to the project core. Although some of these spikes occur simultaneously in all three participant classes, e.g., in mid-June, 2015, we observe others that only concern the activity of 1 or 2 classes. For instance, we observe a considerable increase in communication from core members to unclassified members around the middle of July in 2015. Simultaneously, the communication originating from unclassified members to core members spikes and one week later we see an additional, albeit smaller, upturn in unclassified-unclassified interaction. The communication originating from peripheral developers does not exhibit any significant changes in this time period.

In addition, we investigate the class interaction in a project with a very regular release cycle – Chromium. The relative and absolute class interaction in Chromium are presented in Figure 4.17 and 4.18, respectively. We observe that core-core communication generally amounts for 40 % to 70 % of core activity including some fluctuations. This value is of similar stability, albeit slightly lower, when considering peripheral-core and unclassified-core communication. Generally, uprises in the preference for core members as communication partners seem to affect all three groups simultaneously, e.g., the time period before the release of version “45”. We do not observe any general tendencies of the core-core communication in the context of the release cycle. In some cases, e.g., the time period leading up to version “44”, a noticeable decrease can be observed which mainly relates to the core-core interaction, affecting peripheral-core communication and unclassified-core communication less strongly and not significantly, respectively. In other release cycles, such as the time period between version “46” and “47”, we observe a considerable increase in the preference of core members as communication partners in all three participant classes. We do not observe different tendencies when considering the absolute class interactions shown in Figure 4.17, although it seems as though the uprises in the relative core-core communication manifest even more significantly in the corresponding absolute value.

In our results concerning Qemu, shown in Figure 4.19, we observe the division of the project history into two distinct phases. In fact, these phases correspond exactly to the phases of relative class activity which we presented in Section 4.10. Within a phase, the group-specific preferences for communication partners are only subject to small variations. Thus, the communication preferences seem stable for the whole project history or at least within the general project phases, which is in line with the results we observed for our other subject projects.

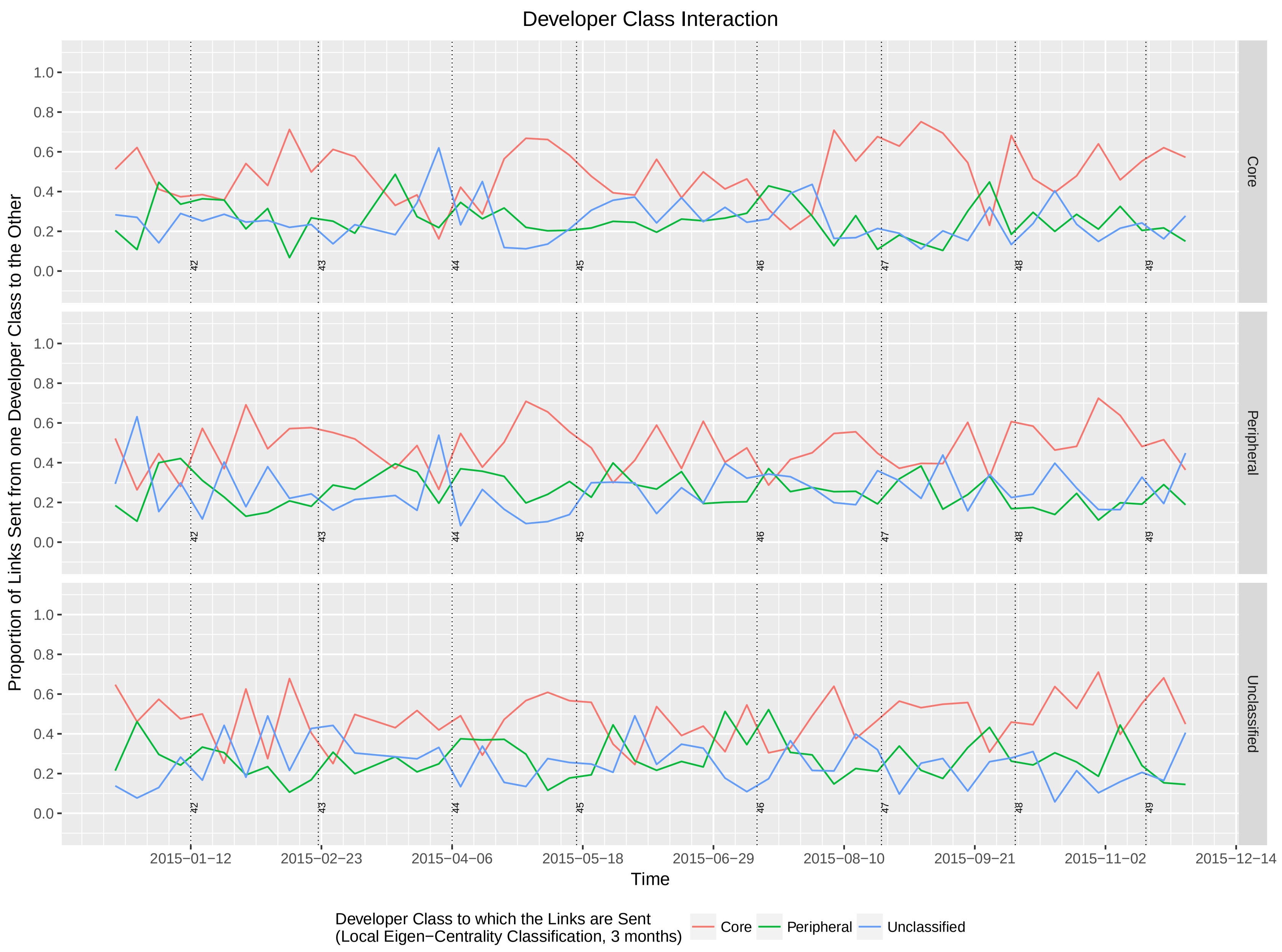

Fig. 4.17 Relative Edge Class Frequency Evolution in Chromium

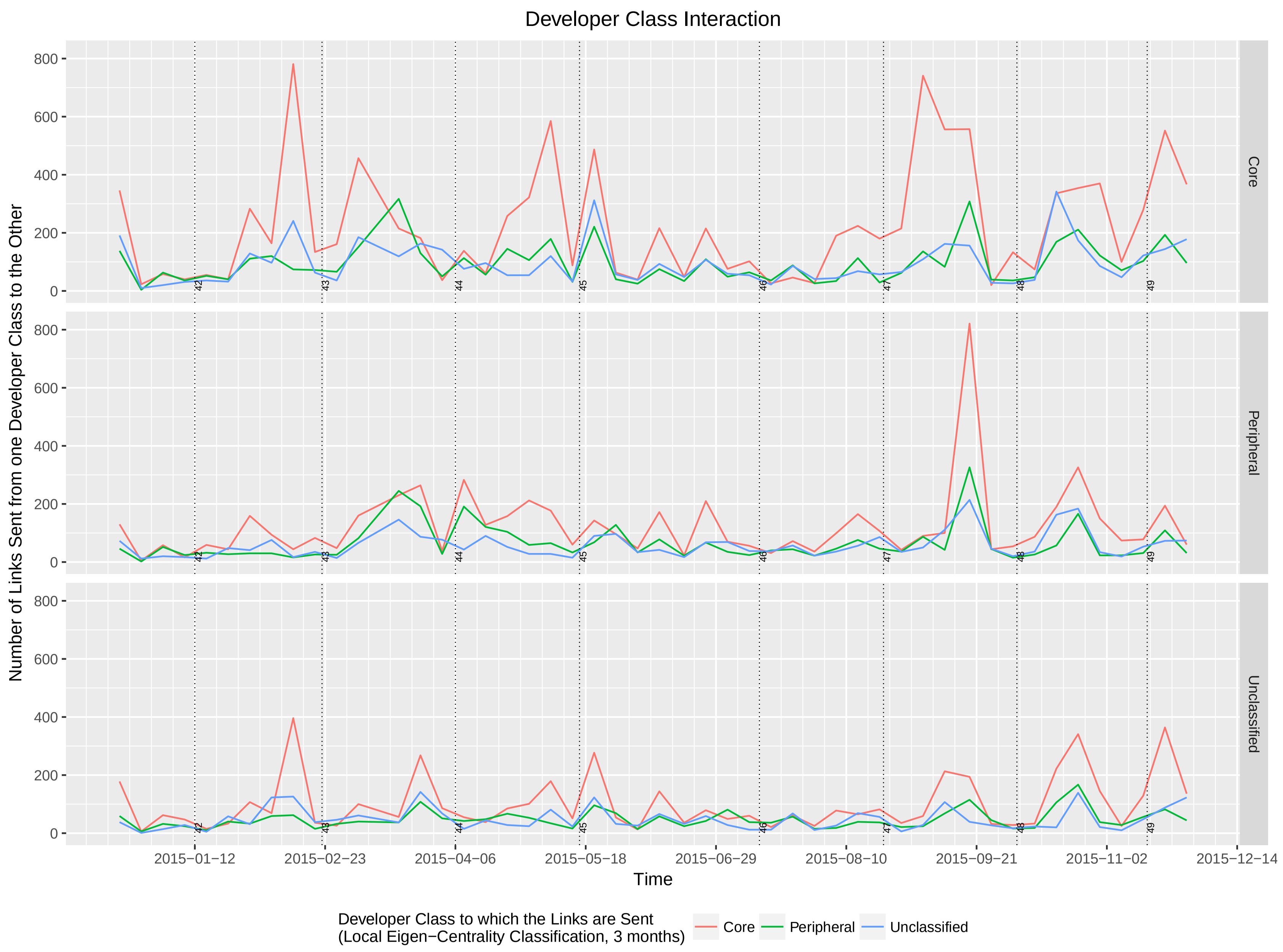

Fig. 4.18 Absolute Edge Class Frequency Evolution in Chromium

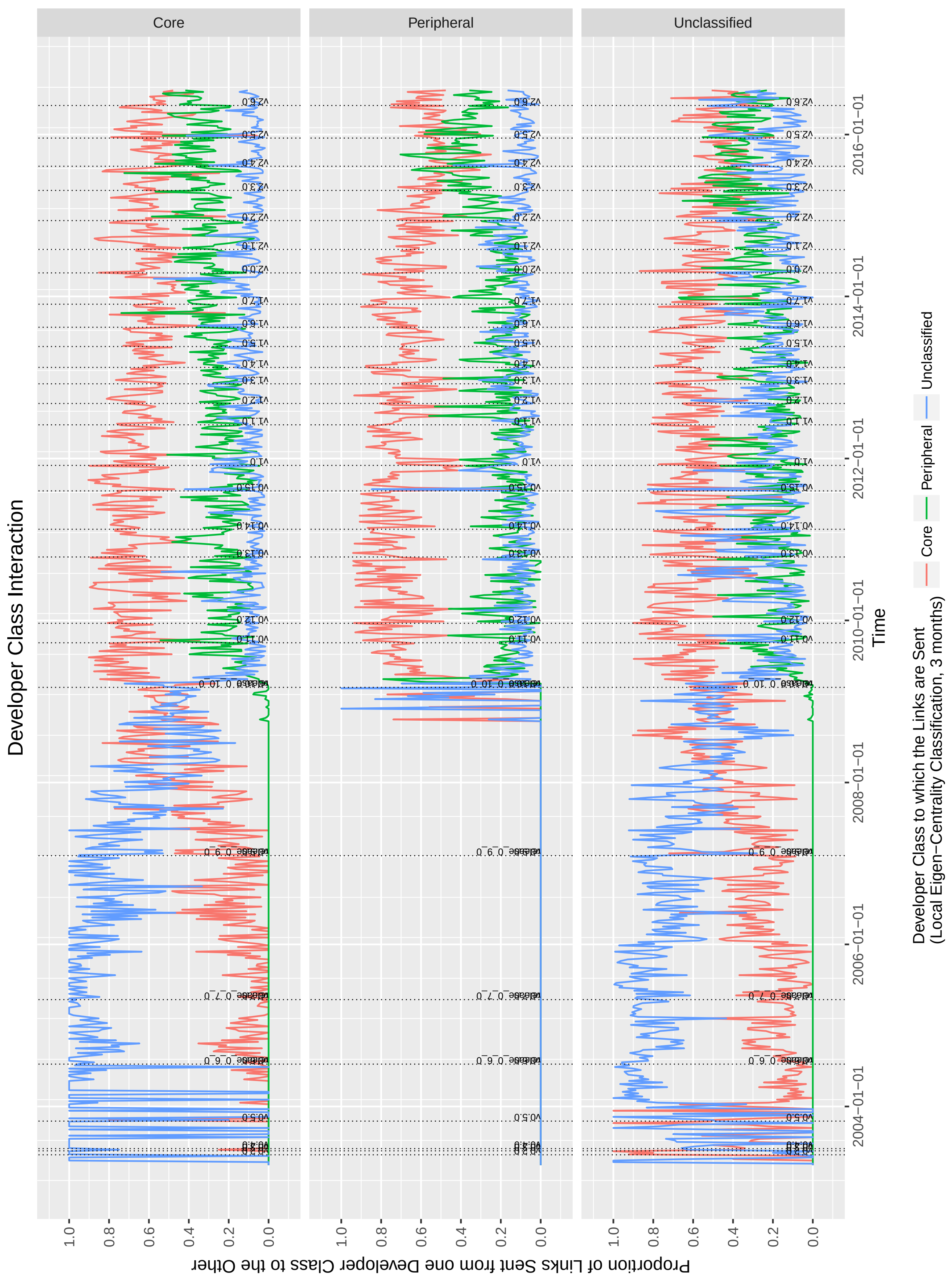

Fig. 4.19 Relative Edge Class Frequency Evolution in QEMU

Overall, we did not observe any release-specific patterns in the class interaction. We observe a compartively high stability in the group-specific communication-partner preferences either for the entire observed project history or for the general project phases. Although we did see core-specific changes in the preference of core members as communication partners, these did not usually correspond to specific points in the release cycle. In some cases, we observed increases in core-core communication before releases but these seem like random fluctuation instead of consistent changes in activity. We draw this conclusion because the phenomenon occurred rather seldomly, we could not observe it consistenly in any single project, and there were generally as many release cycles where we could observe the inverse effect, i.e., a decrease in core-core communication prior to a release. These results do not change qualitatively when considering the absolute numbers as activity indicator. All things considered, we **reject H**3.5.

5 Discussion

We begin this section by discussing our findings. In Section 5.1, we break our results down by the research question they pertain to. Furthermore, we disclose possible threats to the internal as well as external validity of our study in Section 5.2.

5.1 Discussion of our Findings

In the following, we discuss our findings regarding the three research questions we defined in Section 3.1.

5.1.1 Research Question 1

In **RQ1**, we investigate with developers from which group core, peripheral, and unclassified project members communicate primarily. Our hypotheses concerning this research question are the following:

H1.1: Peripheral developers as well as core developers prefer core contributors as communication partners.	**[Accepted]**
H1.2: In a majority of e-mail threads, at least one core developer participates.	**[Accepted]**
H1.3: In a typical e-mail thread, no more than three different peripheral developers participate.	**[Accepted]**

Our results support previous findings [16] which indicate considerable size differences between the project core and the periphery. In no subject project, did we observe that the project core made up more than a quarter of mailing-list contributors. Surprisingly – considering we investigate the primary *development* mailing list whose proclaimed purpose is the discussion of development issues between code contributors [11] – all subject projects were characterised by a very large proportion of non-developers as mailing-list contributors. This proportion ranges from circa 40 % in Wine to an extremely high 90 % in OpenSSL. These results fit in with prior research which indicated that development mailing lists do not primarily discuss matters pertaining to the software development [11].

The involvement of peripheral developers on the mailing list seems to be very project-specific as we observe some projects with almost no peripheral developers in the mailing-list community, whereas the periphery makes up more than a third of the mailing-list contributors in others. The level of involvement of the periphery seems to be linked to the communication guidelines of the project in question, i.e., whether the mailing-list discussions are supplemented with other channels of communication. In projects which rely on issue trackers or features of the hosting platform, e.g., Django, we observe comparatively little peripheral involvement on the mailing list, although the core involvement on the mailing list seems relatively unaffected by this phenomenon. This could indicate that core members will be active on the mailing list in any case, e.g., in their form as coordinators as well as making announcements, but that the discussion of implementation details, in which the periphery is usually involved, is moved to other channels of communication.

We observe that all contributor classes in all casestudies show a clear preference for core members as communication partners. This finding supports prior research which indicated that the core is the project's leadership structure which takes on a coordinating role in the developer community [17, 8]. In addition, this pronounced preference indicates that core members are valued by all groups in the developer community, e.g., due to their possible role as advice-givers and project coordinators. Additionally, peripheral and unclassified members might communicate with core developers to affect future project decisions over which the core exerts considerable influence [16]. Generally, we observe that the project core is not only characterised by a higher level of activity regarding source-code contributions, but also seems to show structural differences in their communication due to their role. Our findings could be an indicator that all participant groups are conscious of the status and role of core members and, thus, prefer them as communication partners.

Moreover, we observe that core members have the possibility to shape the project communication considerably, as they are present in almost all threaded mailing-list discussions: Threads without any core involvement are very rare amounting for less than a third of communication in ten of our eleven subject projects. This is especially notable due to the little absolute number of core contributors. This finding supports the assumption that core members are valued advice-givers and that – due to their role in the project leadership structure and their extensive knowledge regarding the project architecture [16, 8] – their opinion is needed or helpful in most project discussions. The one project which exhibits considerably less core involvement in the mailing-list threads, OpenSSL, might differ so strongly because there are extremely few core members on the mailing list. In fact, it is notable that, given that only 5% of mailing list contributors are part of the project core, most threads still contain at least one core participant. This could be interpreted as an indicator for the great importance of core developers on the mailing list.

As we seldomly observe more than one core developer in a thread, we hypothesise that this might be due to the fact that separate, possibly private communication channels are used for communication among core members.

Additionally, this tendency might indicate that one core developer is usually sufficient for giving advice and coordinating in a ML-thread. In all subject projects, we find that more than three quarters of threads contain two core developers or less. However, this effect might be partly explained by the short threads (see Figure A.4) and especially by the small number of developers per thread (see Figure A.3).

Peripheral involvement in the threaded discussions, however, is very limited as peripheral developers only contribute to less than a third of mailing-list threads in all subject projects. Even in the cases in which a the periphery does participate in a thread, it happens only very rarely that more than two members of the periphery contribute to the same thread. This might indicate that peripheral members often use threads to ask questions which are, then, answered by core members. In general, it seems as though there is little discussion among the members of the periphery. One possible explanation for this phenomenon is the fact that peripheral developers usually do not possess extensive knowledge regarding the general system architecture and, thus, cannot contribute much to implementation-related discussions.

The non-developers' level of involvement on the mailing list varies greatly between different casestudies. We find that, in some projects, unclassified contributors significantly mark the project communication, e.g., OpenSSL, in which a majority of threads contains at least three different non-developers. In others, such as Ffmpeg, three quarters of threads are completely without non-developer participation and even in cases of unclassified involvement, we seldomly observe multiple non-developers in one thread. Similar to the peripheral involvement, tendencies seem to be strongly linked to the communication guidelines in the channel.

As in our example above, we observe that Wine – a project with rather traditional communication guidelines which mainly uses the mailing list to discuss development-related matters and patches – is characterised by comparatively little involvement of non-developers on the mailing list. In this particular case, one reason for the relatively small proportion of non-developers might be that end users are explicitly encouraged to use other forums provided on the Wine-website [32]. In Django's mailing list, we observe a comparatively high involvement of non-developers. Investigating possible reasons for this phenomenon, we find that, although users are discouraged from posting support questions to the mailing list, there is no mailing list or forum dedicated to feature requests and bug reports and, thus, the development mailing list is the primary communication channel to which these issues are posted [24].

Additionally, the proportion of non-developers might not only indicate the general orientation of the mailing list, but could also be an indicator of the treatment of mailing-list-only contributors. For instance, a very small number of non-developers in threads might indicate that the project follows very strict commnication guidelines and non-developers receive no answers to their postings because they post to the wrong list and, thus, these unanswered threads are filtered out in our analysis.

Interestingly, the developer classes' proportions, i.e., the relative frequency of the classes, seem to affect the projects beyond the mere composition of the developer community: we observe structural differences between the communication in projects with comparatively little proportions of unclassified contributors and those in which the mailing-list community contains a great number of non-developers. In OpenSSL, which is characterised a very low proportion of developers, the preferences for core members as well as peripheral members are higher than in other projects. Similarly, we observe a comparatively little preference for core members in Wine – a project whose mailing-list community contains quite a large proportion of core members. This projects exhibits a core preferences that are only half as strong as the median core preferences of all projects. Thus, it seems as though the preference for core members is especially pronounced when there are only few core members on the mailing list. Based on our data, we cannot draw any conclusions of whether this is due to a higher level in activity of these few core members, to the scarcity of members with project expertise, or to an effect we have not yet considered.

5.1.2 Research Question 2

In **RQ2**, we pose the question of how core-core, core-peripheral, and peripheral-peripheral communication episodes differ in longevity and intensity. We formulated the following hypotheses:

H2.1a: Communication between a core developer and a peripheral developer tends to be short-lived.	**[Rejected]**
H2.1b: Core-peripheral communication may be intensive.	**[Rejected]**
H2.2a: Communication among core developers is comparatively long-lived.	**[Accepted]**
H2.2b: Core-core communication may vary in intensity.	**[Rejected]**
H2.3: Communication among peripheral developers is rather short-lived.	**[Accepted]**

In general, we observe surprisingly high communication longevities: When considering only the communication episodes with multiple interactions, i.e., the episodes for which a real longevity could be computed, the typical communication episode has a longevity of more than 1 week in all eleven casestudies. In ApacheHTTP, we even find a typical communication longevity of more than 1 month. However, within projects, we see very pronounced differences in the communication between different classes. This serves as another indicator that there are structural differences between the participant classes. In addition, it might indicate that one developer might adapt his/her behaviour to the class of his/her communication partner.

When comparing core-peripheral communication to the overall project communication, we do not generally find the former to be of smaller longevity or higher intensity than the latter. In most cases, we observe quite a strong inverse effect. However, there are some subject projects which support our assumptions. These results do not change qualitatively when considering only communication episodes with multiple interactions or using a different intensity metric. Interestingly, we do not see any universal tendencies regarding core-peripheral communication longevity. The effects we do observe are very project-specific and, hence, might indicate that the involvement and treatment of peripheral developers varies greatly between different OSS projects. However, our assumptions hold when comparing the core-peripheral communication to the core-core and peripheral-peripheral communication as explained below.

These results can be partly explained by the fact that we underestimated the significance of unclassified developers in the mailing list. In fact, it seems natural that all communication episodes among code contributors would be of greater longevity than communication among non-developers due to the fact that non-developers will not usually discuss implementation details and are possibly not as involved in long-lived discussions concerning parts of the system architecture as well as the project's future. As the project core and periphery might have a stronger link to the project due to their personal involvement in the project in the form of code contributions, we would expect their participation in the mailing list to be longer than that of non-developers. This is one possible factor that could explain why all communication among developers is comparatively long-lived.

In addition, we find that communication among core developers is markedly more long-lived than the general project communication. This phenomenon can be observed regardless of whether a filtering of multi-interaction episodes is applied. The effect is visible regarding typical communication episodes, rather short-lived episodes, and – to some extent – longer episodes, i.e., the 0.5, 0.25, and 0.75 quantile, respectively. However, it is especially pronounced consdering the 0.25 quantile. Based on these results, we draw the conclusion that the typical minimum duration of a core-core communication episode greatly exceeds that of general communication episodes.

One possible explanation is that a pair of core members who communicates once may either be involved in a general discussion regarding the project, e.g., future plans, which may last for a relatively long time, or they might work on overlapping parts of the system. Due to the longterm involvement of core members it seems likely that they each have their areas of expertise and certain parts of the software on which they concentrate. Thus, if two core members coordinate their efforts once and their responsibilities typically do not change, this would explain that this pair of developers needs to communicate again and again over the project history as their respective parts of the system are most likely linked and they need to coordinate before and during possible changes in the code and architecture.

Peripheral-peripheral communication episodes are generally characterised by their comparative brevity. It seems as though communication among members of the periphery is especially strongly affected by communication episodes with only one interaction. Nonetheless, even when filtering out these short episodes, peripheral-peripheral communication is significantly shorter than the general project communication. This may be partly explained by the little proportion of threads in which multiple members of the periphery participate (see Section 5.1.1). In addition, it seems likely that members of the periphery do not need to coordinate over a long period of time as they typically do not contribute code for long time spans. In addition, they usually do not change big parts of the system architecture and, thus, their work requires little coordination. In the cases in which peripheral developers need coordination and guidance concerning their contributions, it seems more likely that they approach knowledgeable core members than other peripheral developers.

We gain additional insights into the behavioural differences between the two developer classes – the core and the periphery – when comparing the different kinds of communication among developers. Most interestingly, we find that core-core communication episodes are generally more long-lived than core-peripheral communication which is, in turn, of greater longevity than peripheral-peripheral communication. Thus, although not all the assumptions we formulated in **RQ2** hold when comparing communication among developers to the general project communication, we observe that they are supported by our results when filtering out communication in which non-developers participate. These findings indicate that – regardless of the project-specific involvement of ML-only contributors – the structural difference as well as the hierarchy between the project core and periphery seem to remain stable in the mailing-list communication.

Considering the communication intensity, we observe two contrary tendencies: In two thirds of our subject projects, peripheral-peripheral communication episodes are the most intensive, whereas, in the remaining casestudies, core-core communication is of the greatest intensity. These results do not change qualitatively when using mails instead of links as metric of intensity. Interestingly, these results not only concern the typical communication episodes but are consistently reflected in the distributions of core-core, core-peripheral, and peripheral-peripheral communication intensity. Hence, it seems as though there are two classes of projects, which differ greatly in the nature of the peripheral mailing-list involvement.

The possibility that core developers communicate in a more concise manner and – opposed to peripheral members – do not typically need repeated, iterative advice on implementation decisions, could be one reason for the comparatively little intensity of core-core communication in most subject projects. Peripheral-peripheral communication episodes may be relatively intensive as the participation of peripheral developers may be marked by relatively short and irregular, but intensive bouts of activity which are reflected in the communication among members of this group. In addition, we deem it likely that peripheral developers, which typically do not hold roles of coordinating other developers' contributions, communicate primarily when they work on parts of the system that are closely linked. Thus, the rare occasions when pairs of peripheral developers do communicate, they communicate intensively as they adapt their respective changes in the source code using repeated feedback from each other.

In addition, we found that, in a majority of projects, variance in the intensity of core-core communication episodes is significantly smaller than not only the variance in total-total intensity but also than the corresponding value for the core-peripheral as well as the peripheral-peripheral communication episodes. Interestingly, the cases where the core-core communication seems more varied are those cases which showed anomalies concerning the general distribution of the core-core communication compared to the project-wide values. Thus, this supports above-mentioned thesis that few casestudies are characterised by structurally different patterns in the communication among their developers.

Although there are a few exceptions to this rule in which core-core intensity variance exceeds that of total-total communication, we observe relatively little variation in the core-core communication intensity in a majority of our casestudies. This comparatively small variation might be an indicator of a possible homogeneity of the core group: As core members are characterised by a prolonged involvement in the OSS project, they likely adapt to project norms regarding not only the code contribution but also the communication with other project members. This would set them apart from the periphery and especially the unclassified members and is a possible explanation as to why interactions among them are marked by a greater uniformity than core-peripheral, peripheral-peripheral, or general contributor interactions.

Moreover, the fact that inter-project differences in typical communication longevity manifest less strongly in core-core communication than in the other forms of communication may indicate that a certain uniformity not only exists concerning the core developers within a project, but even across different OSS projects. Possible explanations for this phenomenon are their general role in the project leadership structure and as coordinators [16], which favour – or even necessitate – certain behaviours when communicating.

5.1.3 Research Question 3

In **RQ3**, we investigate to what extent the three developer classes contribute to the total communication activity. In addition, we analyse whether we can observe any changes over the course of a release cycle. We analyse the following hypotheses:

H3.1: Code contributors are more active regarding the average number of mails written than mailing-list-only contributors.	**[Accepted]**
H3.2a: Core developers write a majority of e-mails.	**[Rejected]**
H3.2b: The project core initiates more than 50 % of mailing-list threads.	**[Rejected]**
H3.3: The number of e-mails written by core developers increases significantly prior to a release.	**[Rejected]**
H3.4: Peripheral communication activity spikes during the beginning and end phase of the release cycle.	**[Rejected]**
H3.5: Core-core communication increases in the time period prior to a release.	**[Rejected]**

We observe that the activity level measured via code contributions is strongly reflected in the mailing-list activity. We find a universal tendency for core members to be more active than peripheral members with regards to mails as well as to the initiation of threads.

Considering the activity of an average class member, our findings support the existence of a strict hierarchy of activity in which core members dominate peripheral members and in which non-developers are the least active contributors. This result is visible in all subject projects when considering mails as measure of activity. Even when considering the number of threads initiated, we find this same hierarchy in all but one casestudies. Hence, we observe that – as indicated by previous findings [1] – source-code activity and mailing-list contributions are strongly correlated.

When investigating the proportion of mails as well as threads that each class contributes, we observe slightly different results. We find that the core is universally the most active contributor class. However, core developers are usually not responsible for more than 50 % of messages posted to the mailing list. This finding can be explained via the small number of core developers. Considering that the proportion of core members is always smaller than 25 % in our casestudies – mostly around 15 % – these high proportions of core mails are actually another indicator for the great activity level and importance of the project core in the mailing-list community.

Using the initiation of mailing-list threads as activity measure, we observe very different results: In a majority of our casestudies, non-developers initiate more threads than the project core and the periphery. In six of our eleven casestudies, ML-only contributors even initiate the bulk of threaded mailing-list discussions. Nonetheless, some cases remain where core developers significantly dominate the other developer classes with regard to the number of threads inititated. The cases of strong core thread initiation seem to be linked, not only to the proportion of core members on the mailing list – as in the example of Wine – but also to the orientation of the mailing list. One example for the latter is Ffmpeg which explicitly directs users to a user-specific mailing list and provides a separate list for bug reports [9]. Thus, the development mailing list is intended exclusively for matters of development and the submission of patches. It is possible that misplaced messages by non-developers do not receive answers and are, thus, filtered out for our analyses, explaining why the remaining threads are, for the most part, initiated by developers and particularly core members. Therefore, whether core members or non-developers dominate the mailing list with regard to the number of threads initiated can possibly serve as an indicator for the orientation and actual usage of the mailing list.

In addition to the aggregated data we considered in **RQ1** and **RQ2**, we adopt an evolutionary perspective in the investigation of **RQ3**. Interestingly, we find that the classes' proportion of contributions as well as the specific preferences of different groups as communication partners seem to be project-specific, differing considerably among our casestudies, but remaining very stable within projects. Although we observe two projects in which this stable phase only comes into play after a certain time or is lost at one specific point in time, we generally find that – within phases which last more than 1.5 years in our subject projects – these stable phases exist in all our subject projects. Thus, the interaction preferences and the levels of activity seem to be characteristic of the specific OSS project. Interestingly, the changing of phases seems to be linked to releases, perhaps due to explicit changes, e.g., in the OSS project organisation or the communication guidelines.

We do not observe any universal, release-specific patterns. This might be due to the release systems in our eleven subject projects varying too greatly. Nonetheless, in most projects, our data does not indicate any recurring tendencies regarding activity or interaction patterns which could be linked to releases. Although some small, recurring fluctuations in core preferences can be observed, these do not seem to be regular in the context of the release cycle.

However, in several projects, we find a semi-regular, recurring rise in core activity about 2 weeks to 4 weeks after a major release. It seems likely that this phenomenon is linked to a coordination phase in which core developers plan for the next release, e.g., considering which features out of a multitude of suggestions might be included in the next version and setting a general schedule for the next release. In addition, we observe a decline in the core communication activity towards the end of a release cycle in several cases. One possible explanation of this phenomenon is the assumption that less coordination is necessary towards the end of a release cycle. For instance, most projects perform a *feature freeze* at one point in the release cycle after which no more features are added to the current version. After this feature freeze, only bugfixes are applied but the integration of new features which is likely linked to extensive discussions involving the project core, is finished. Nonetheless, these two effects only occur in some release cycles and do not seem like a universal phenomenon. Thus, the assumption that they are manifestations of how the release cycles affect the project communication seems unconvincing.

We observe the strongest tendencies of such, possibly release-specific, effects in projects with very rigid, regular release schedules, such as CHROMIUM.

In this particular project, we observe two phenomena, one of which affects all groups in equal measure and one which mainly shapes peripheral activity. However, these effects do not correspond to general changes in phases of the cycle but only affect two very specific points in the course of a release cycle.

All things considered, our data is not sufficient to substantiate any claims of release-specific, regular changes in activity and interaction patterns. Although we observe some single cases where the communication may be affected by certain points in the release cycle, the general evidence does not seem convincing.

5.2 Threats to Validity

In this section we discuss threats to the internal and external validity of our empirical study.

5.2.1 Internal Validity

The threats to the internal validity of our study mainly relate to the choice of our independent variables.

The most significant possible problem might be our choice of data sources. We analyse the primary development mailing list because it constitutes an archive of the project communication. The mailing list has historically been considered the hub of project communication [8, 11] and is generally where newcomers' first interaction with the project community takes place [13]. Therefore, it should still be of considerable importance to the project communication even in projects that supplement the mailing list with other communication channels.

Nonetheless, most projects have several mailing lists for different purposes [16]. We choose to analyse the primary development mailing list as we want to study the communication and coordination of the code contributors on the mailing list which should traditionally take place on the primary development mailing list. However, there may be several development mailing lists, e.g., one mailing list for patches and one exclusively for core members regarding the planning of releases, on which developers communicate. Even when considering projects with strong mailing lists that follow more traditional communication guidelines, core developers may communicate over private channels. Nevertheless, since we mainly investigate the interaction between core and peripheral developers, this phenomenon should not substantially distort our findings.

A further possible problem concerning the choice of the primary development mailing list as data source is that its actual usage may not reflect its declared intent. Prior research has shown that a great proportion of messages posted to the development mailing list may not actually pertain to the project development [11]. We mitigate this risk by excluding threads which did not receive any answers from our thread-level analyses as these most likely did not consist of veritable developer communication but automated or misplaced messages.

The second important problem concerns our choice of classification metric. We choose to classify developers based on VCS-data. As the VCS is an integral part of the OSS software-engineering process, its data should not be significantly corrupt [16]. We operationalise the notion of core and periphery – which has been extensively analysed in prior research, e.g. [16, 17, 8, 29, 10] – via the network-based eigencentrality metric. Other research has shown that this approach accurately reflects developers' perception of project roles [16]. Moreover, we additionally performed all analyses using a commit-count-based classification and did not generally observe significant differences compared to our findings.

Our choice of three-month windows for the local classifications is well-founded as prior research has indicated that networks constructed using this time window accurately reflect developers' perception of the community without obfuscating temporal details [16, 17]. We mitigate the risk that some core developers no longer provide source-code contributions but hold roles as coordinators and advisors on the mailing-list by including longterm mailing-list-core members in the core group. This term designates the developers that are classified as core based on the mailing-list data over a timespan of at least half the analysed project history or alternatively three years in total. Some source-code core developers may not contribute to the mailing list, but – as prior research has shown that core developers usually hold roles as coordinators in the projects instead of merely being the most active code contributors [8] – their effect on our findings should be negligible.

The third threat to the internal validity of our study concerns our construction and analysis of communication episodes. The most significant possible problem is the choice of 7 days as communication window threshold. Although we choose this value based on the temporal distances between subsequent messages of e-mail threads (as shown in Figure A.5), this value might not be an accurate indicator of the typical temporal distances between *pair-wise* communication. In addition, we could have chosen project-specific thresholds to guarantee more fine-granular results. This is an important aspect to consider for future work. However, as the temporal distances did not majorly differ among our 11 subject projects, our chosen threshold should not greatly bias our results.

In addition, we consider communication episodes spanning multiple threads. Although developer interactions over multiple threads may not necessarily pertain to the same topics, they link the same developers and are both part of the communication between these pairs of developers. Thus, our choice of multi-thread communication episodes is intentional as we do not want to perform a thread-level abstraction of developer communication. In addition, measures of longevity would be highly skewed when considering interactions for different threads separately, as the typical threads are not of great longevity (see Figure A.4). Hence, this choice fits our research questions better than performing thread-level abstractions.

Misplaced messages, e.g., user questions posted to the wrong mailing list, as well as automated messages, e.g., by the continous build system, could theoretically distort our results. However, as we apply a filtering to the threads removing all those that did not receive any answers – a condition which should apply to misplaced as well as automated messages – we expect their influence on our findings to be negligible.

Our findings might be biased – especially concerning communication among peripheral members – by the great number of communication episodes with only one interaction which distorts our measure of longevity. We mitigate this risk by separately analysing only communication episodes which contain more than one interaction and comparing these results to our findings. In general, this filtering process only quantitatively alters our results while the tendencies we observe, i.e., the qualitative findings, remain stable. We employ a similar approach to guarantee that our measure of intensity is valid: As one e-mail in a very long mailing-list thread may produce several edges between a pair of developers and, hence, distort our results of communication intensities, we compare our results considering link-based intensity with the corresponding data using a mail-based intensity metric.

Another possible way in which especially long mailing-list threads may bias our results is our network construction mechanism. We do not use any form of communication decay, i.e., we consider that two participants in thread interact even if their respective contributions are several weeks apart. Although this is admittedly a threat to the validity of our findings, its effect should not be significant as most threaded mailing-list discussions in our subject projects are very short (see Figure A.4) and subsequent contributions to a thread are seldomly more than 24 hours apart (see Figure A.5). Thus, our assumption that even the last message in a thread represents an interaction of its author with the initiator of the thread generally seems justified in the projects we investigate.

Undoubtedly, temporary fluctuations may distort our findings. We mitigate this threat by analysing at least 1 year of activity for each subject project. In addition, we analyse more than 13 years of activity for 4 subject projects – ApacheHTTP, Busybox, OpenSSL, and Qemu – which did not generally produce different results than our casestudies with shorter analysed histories. Thus, the distortion of our results by temporary fluctuations should be insignificant.

5.2.2 External Validity

The main threat to the external validity of our findings is the problem that our subject projects might not be representative of the entirety of OSS projects. We only choose relatively established projects with at least one year of development and mailing-list history as we would otherwise not be able to rule out that our results are produced by temporary fluctuations. In addition, we only analyse OSS projects with comparatively large developer communities as outliers and single events in small developer communities might skew the results for the whole project due to the small number of developers and, thus, interactions. Thus, our results may not be relevant for less mature or very small OSS projects.

Prior research indicates that OSS may not be seen as one universal phenomenon but that considerable differences between the projects exist [7]. Hence, the findings we achieved for our eleven casestudies may not be representative of the entirety of OSS development. We mitigate this problem by choosing a very diverse set of 11 casestudies. As shown in Section 3.4, the projects vary considerably in size, domain, technology, source-code activity, mailing-list activity, version control system, release system, and the communication guidelines on the mailing list. Thus, our findings should not be significantly biased.

6 Conclusion

In the following, we conclude our work by summarising our study design and our findings as well as presenting suggestions for future work and possible enhancements of our study.

6.1 Summary

In this thesis, we conducted a long-term, empirical study of the mailing-list communication in eleven OSS projects. We focused on investigating how source-code activity is reflected in the interactions on the project's primary development mailing list. We used a network-based eigencentrality metric to classify developers into core and periphery based on their contributions to and specifically their collaboration in the project's source code. Using these developer roles, we analysed how the differences in the project core, the periphery, and the non-developers manifest in their activity level as well as the characteristics of their communication in the mailing list on the basis of e-mail social networks. We used not only an aggregation of the entire project-history data but additionally provided a fine-granular, evolutionary perspective of the interactions on the mailing list.

With regards to the composition of the mailing-list community, we observed considerable size differences between the classes. Most notably, only a very small proportion of mailing-list contributors was classified as core members. In all subject projects, surprisingly large numbers of non-developers participated even though we analysed *development* mailing lists. The involvement of peripheral developers in the mailing list seemed to be very project-specific and was apparently strongly linked to whether other communication channels were used in the project. In general, we observed pronounced differences between the developer groups which not only pertained to their activity levels but also strongly affected their preferences regarding communication partners and the characteristics of the communication episodes among them.

Considering the communication-activity levels of the different developer groups, we found a strict hierarchy of the activity of an average member indicating that core members dominated peripheral members which, in turn, exhibited higher levels of activity than non-developers. This hierarchy was very pronounced when using mails as indicator of activity and still visible when considering the number of threads started as measure of activity. In addition, the proportion of mails written by the project core exceeded the corresponding values of the periphery as well as of the non-developers. However, there seemed to exist two classes of projects regarding the initiation of threads: In a majority of our casestudies, ML-only contributors were the most active group with regards to threads and initiated the bulk of the threaded mailing-list discussions. Nonetheless, in other projects, the project core surpassed the other contributor groups concerning the inititation of threads. This phenomenon was possibly linked to the orientation of the mailing list, e.g., whether end users are encouraged to participate in the form of feature requests and bug reports.

In addition, core members were present in almost all threaded mailing-list discussions which might be an indicator of their role as coordinators and advisors. However, we rarely observed a thread with more than one core participant which supports the assumption that core-core communication might take place on private channels. Peripheral involvement in the mailing-list threads was extremely limited. Interestingly, we found that the involvement of non-developers was very project-specific. In some cases, we observed that ML-only contributors could significantly shape the project communication as multiple non-developers were present in a majority of threads. Similar to the composition of the mailing-list community, this phenomenon seemed to be correlated to the orientation of the development mailing list and the existence of other communication channels.

Above-mentioned differences manifested not only in the composition of the community and the classes' activity, but also created structural differences in the communication: Although we observed a very pronounced preference for core members as communication partners in all contributor classes in all eleven subject systems, the distinctness of this preference seemed to vary based on the involvement of the three classes. In general, this preference indicated that all contributors were conscious of the role and status which the project core holds.

To investigate pair-wise communication, we introduced the notion of *communication episodes* which correspond to prolonged, connected episodes of communication between two developers, possibly spanning multiple threads. Considering these communication episodes, we found that core-core communication was generally significantly more long-lived than core-peripheral communication which is, in turn, of greater longevity than peripheral-peripheral communication episodes. This phenomenon indicated that, regardless of the project-specific involvement of non-developers, the structural differences between the project core and the periphery remained stable across different projects.

We found two classes of projects based on communication intensity: In most of our subject projects, peripheral-peripheral communication exhibited the highest intensity. Nonetheless, in a few exceptions, core-core communication was of the highest intensity, indicating that communication patterns in different OSS projects vary greatly in the nature of the peripheral mailing-list involvement. In addition, most intra-project and inter-project differences, e.g., in communication longevity, manifested less strongly in core-core communication than in other forms of communication indicating that the core group might be comparatively homogenous and that the interactions among them might share certain characteristics across different projects.

Considering the evolution of the project communication, we found that patterns in activity as well as interaction between the groups, such as preferences for communication partners – although varying greatly across projects – were very stable over time within a project. We observed long-lasting phases during which these characteristics only showed slight fluctuations. Although this stability was lost, respectively found, with one specific release in two of the projects we investigated, it seemed to affect all subject systems. Thus, the classes' interaction as well as their levels of activity seemed to be a project characteristic which was not considerably affected by temporary fluctuations. Although we observed some tendencies that might indicate an influence of the release-cycle on the project communication, we did not find significant recurring effects that could be definitely linked to the releases.

In summary, we showed that – although some characteristics of the project communication seemed to be linked to the orientation of the mailing list – the structural differences and the relation between the project core and the periphery manifested in similar ways in our subject projects. In general, the differences in source-code activity were strongly reflected in the level and nature of mailing-list contributions. In addition, the core group did not only show greater homogeneity within a casestudy but also seemed to share some characteristics across different OSS projects. Interestingly, class activity levels and interaction preferences were project characteristics but stayed consistent within a project during long-lived phases.

6.2 Future Work

It would be a valuable extension of this work to deepen the evolutionary perspective we adopted in **RQ**3. We plan to apply the evolutionary analyses to the class activity level as measured in the number of threads initiated in addition to the mail-based analysis we performed in this work. As we observed that communication patterns generally stay stable over long-lasting phases, the analysis of what prompted changes at specific points in time would be interesting. In addition, one could investigate the two projects in which this stability was lost, respectively found: Questions such as "What prompted these changes? How did they affect the content of the mailing list as well as the quality of source-code contributions?" would give valuable insights into how the communication structure adapts to coordination needs in the project. Additionally, further research into the release-specific patterns we observed in Chromium, particularly a quantitative analysis of these effects, might be interesting. The release-specific analyses could be extended to more OSS projects – especially those with rather rigid release schedules – to investigate whether any other, possibly project-specific, effects linked to the releases can be found.

We plan to give more context to our results by examining not only what proportion of mailing-list contributors belongs to the project core or periphery but to investigate what percentage of the code contributors who are classified as core, respectively peripheral, are actually active on the mailing list. Moreover, in future work, we plan to investigate over how many threads a pair of contributors typically communicates over the course of one communication episode. An extension of our findings regarding communication intensity by using a finer metric of intensity – e.g., one which is not as strongly linked to the duration of the episode but reflects the evolution of the intensity over the course of the communication – could provide valuable insights. Verifying our results by using variable communication-episode time windows, in particular project-specific ones, could show whether our findings are supported when a different perspective vis-à-vis what constitutes ongoing communication is adopted.

In addition, as we have found most threads to contain very few contributors, an analysis of the characteristics as well as of the content og particularly long threds would be interesting: Hypotheses such as "In especially long threads, the proportion of core members is comparatively high." could be investigated. Similarly, work that studies whether there are any pairs of developers that communicate very intensively and to which groups they belong could give valuable insights into the structure of the projects' mailing-list communities. In general, a study of the frequency of interaction for each communicating pair of developers, i.e., in how many communication episodes with one another they are involved, could be interesting.

As more and more projects use different communication channels, it would be valuable to replicate or combine our analyses with bugtracker data as foundation for the construction of the communication networks used in this study. A *market-basket analysis*, i.e., considering frequent subsets instead of only pairs of contributors, would constitute a valuable extension of this thesis, allowing us to perform a group-level abstraction of communication patterns. Lastly, an extension of our work by considering not only the number of mails written but instead quantifying messages' importance by weighing mails according to their size or even using natural language processing to gain insights into the content of the communication would be interesting.

A

Additional Material

In this section, we provide supplementary material and figures illustrating additional results of our study. We only present these results for Qemu. For all material for the other subject projects as well as results of analyses using a commit-count developer classification, the interested reader is referred to the digital copy of this work.

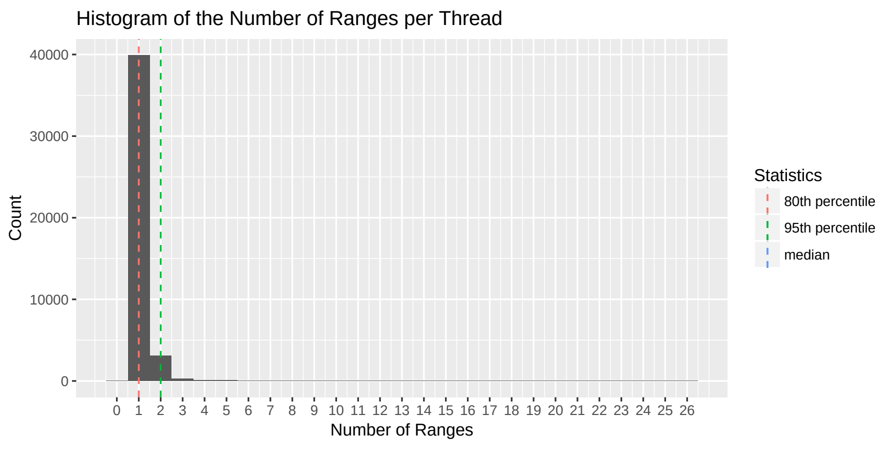

Fig. A.1 Histogram of the Number of Ranges Threads Span in Qemu

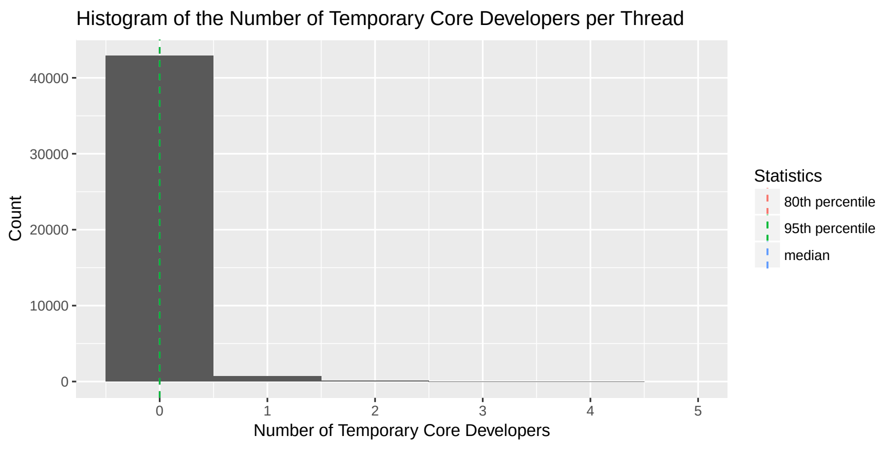

Fig. A.2 Histogram of the Number of Temporary Core Developers in Qemu-Threads

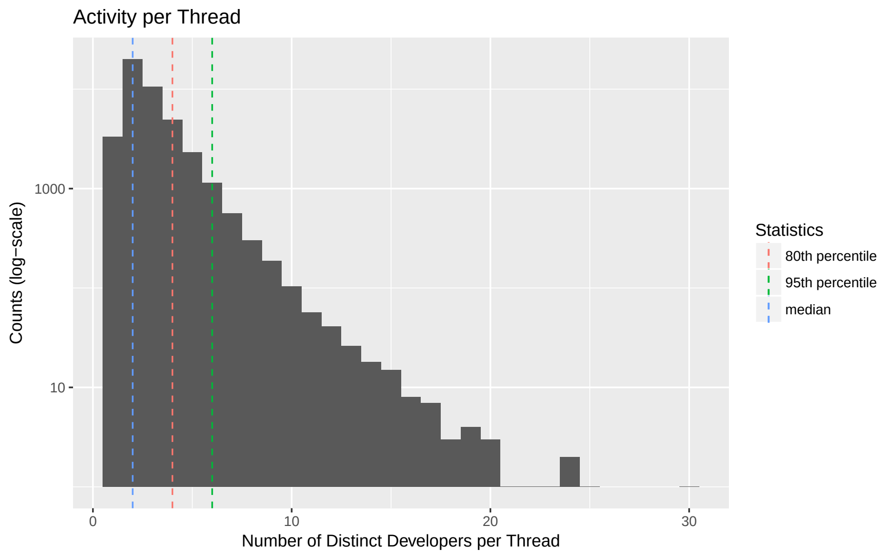

Fig. A.3 Histogram of the Number of Participants in Qemu-Threads (logarithmic scales)

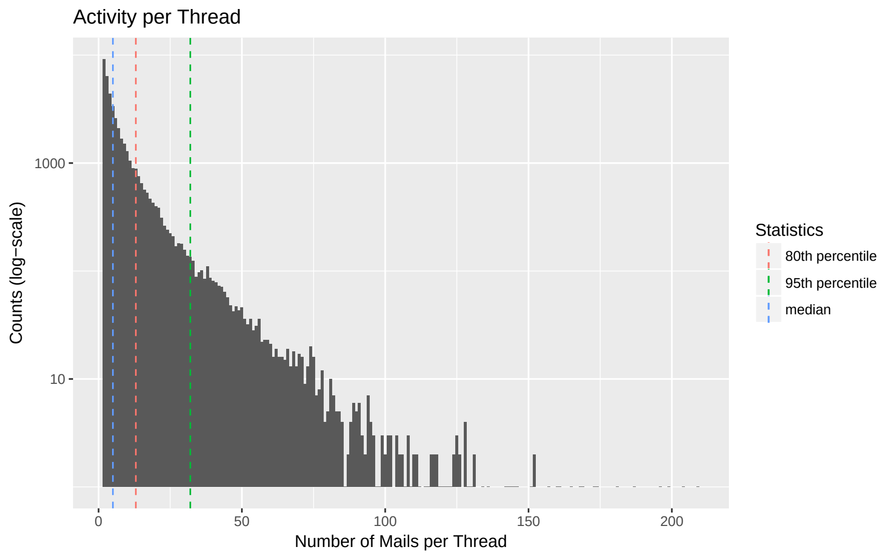

Fig. A.4 Histogram of the Number of Mails in Qemu-Threads (logarithmic scales)

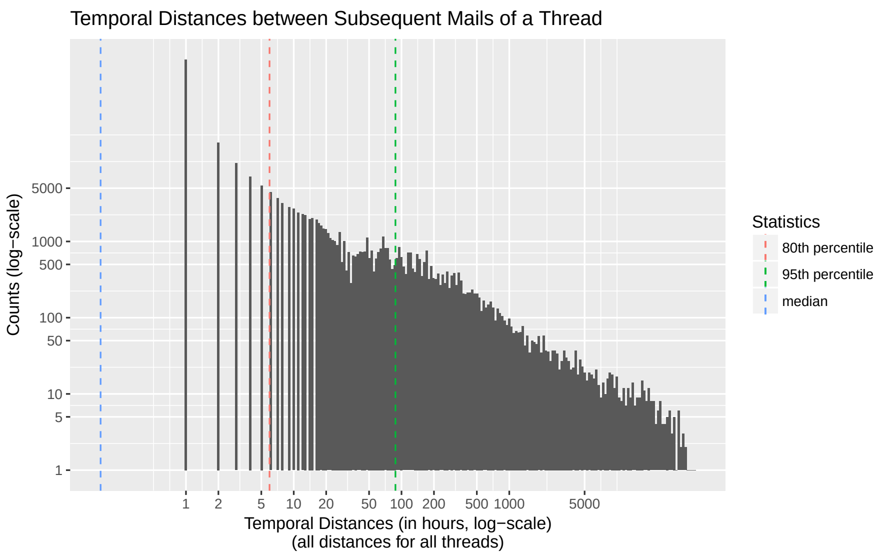

Fig. A.5 Histogram of the Temporal Distances between Subsequent Messages in Qemu-Threads (in hours, logarithmic scales)

B

Bibliography

[1] Christian Bird, Alex Gourley, Premkumar Devanbu, Michael Gertz, and Anand Swaminathan. “Mining Email Social Networks”. In: *International Workshop on Mining Software Repositories (MSR).* 2006, pp. 137–143 (*cited on pp. 1, 4, 5, 6, 8, 14, 15, 18, 29, 31, 80*).

[2] Christian Bird, David Pattison, Raissa D’Souza, Vladimir Filkov, and Premkumar Devanbu. “Latent Social Structure in Open Source Projects”. In: *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE).* 2008, pp. 24–35 (*cited on pp. 3, 4, 5, 13*).

[3] Stephen Borgatti and Martin Everett. “Models of Core/Periphery Structures”. In: *Social Networks* 21.4 (2000), pp. 375–395 (*cited on p. 10*).

[4] Stephen Borgatti, Martin Everett, and Jeffrey Johnson. “Analyzing Social Networks”. Sage Publications Ltd, 2013 (*cited on pp. 5, 6, 10*).

[5] Stephen Borgatti, Ajay Mehra, Daniel Brass, and Giuseppe Labianca. “Network Analysis in the Social Sciences”. In: *Science* 323 (2009), pp. 892–895 (*cited on pp. 5, 8, 10*).

[6] Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. “Who is Going to Mentor Newcomers in Open Source Projects?” In: *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE).* 2012, p. 44 (*cited on pp. 1, 8, 13*).

[7] Kevin Crowston and James Howison. “The Social Structure of Free and Open Source Software Development”. In: *First Monday* 10.2 (2005). ISSN: 13960466 (*cited on pp. 4, 85*).

[8] Kevin Crowston, Kangning Wei, Qing Li, and James Howison. “Core and Periphery in Free/Libre and Open Source Software Team Communications”. In: *Hawaii International International Conference on Systems Science (HICSS).* 2006 (*cited on pp. 5, 10, 11, 19, 28, 29, 74, 82, 83*).

[9] “ffmpeg-devel – FFmpeg Development Discussions and Patches”. 2017-09-10. URL: `https://ffmpeg.org/mailman/listinfo/ffmpeg-devel` (*cited on p. 80*).

[10] Jochen Gläser. “The Social Order of Open Source Software Production”. In: *Handbook of Research on Open Source Software: Technological, Economic, and Social Perspectives.* 4. IGI Global, 2007, pp. 168–182 (*cited on pp. 4, 8, 9, 83*).

[11] Anja Guzzi, Alberto Bacchelli, Michele Lanza, Martin Pinzger, and Arie van Deursen. “Communication in Open Source Software Development Mailing Lists”. In: *Working Conference on Mining Software Repositories (MSR).* 2013, pp. 277–286 (*cited on pp. 4, 5, 13, 21, 73, 82*).

[12] Andrea Hemetsberger and Christian Reinhardt. “Sharing and Creating Knowledge in Open-Source Communities, The Case of KDE”. In: *European Conference on Organizational Knowledge, Learning, and Capabilities.* Innsbruck, Austria, 2004 (*cited on p. 13*).

[13] Carlos Jensen, Scott King, and Victor Kuechler. “Joining Free/Open Source Software Communities: An Analysis of Newbies’ First Interactions on Project Mailing Lists”. In: *Hawaii International Conference on Systems Science (HICSS).* 2011, pp. 1–10 (*cited on pp. 1, 5, 8, 9, 10, 13, 82*).

[14] Chris Jensen and Walt Scacchi. “Role Migration and Advancement Processes in OSSD Projects: A Comparative Case Study”. In: *International Conference on Software Engineering (ICSE).* 2007, pp. 364–374 (*cited on pp. 8, 9*).

[15] Corey Jergensen, Anita Sarma, and Patrick Wagstrom. “The Onion Patch: Migration in Open Source Ecosystems”. In: *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE).* 2011, pp. 70–80 (*cited on p. 12*).

[16] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. “Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics”. In: *International Conference on Software Engineering (ICSE).* 2017, pp. 164–174 (*cited on pp. 6, 7, 8, 9, 10, 11, 14, 15, 17, 18, 19, 28, 29, 31, 33, 73, 74, 79, 82, 83*).

[17] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. “Evolutionary Trends of Developer Coordination: A Network Approach”. In: 22.4 (2017), pp. 2050–2094 (*cited on pp. 1, 4, 6, 8, 9, 10, 11, 12, 19, 28, 29, 30, 74, 83*).

[18] Stefan Koch and Georg Schneider. “Effort, Co-operation and Co-ordination in an Open Source Software Project: GNOME”. In: *Information Systems Journal* 12.1 (2002), pp. 27–42 (*cited on pp. 3, 4*).

[19] Georg von Krogh, Sebastian Spaeth, and Karim Lakhani. “Community, Joining, and Specialization in Open Source Software Innovation: A Case Study”. In: *Research Policy* 32.7 (2003), pp. 1217–1241 (*cited on pp. 11, 12*).

[20] George Kuk. “Strategic Interaction and Knowledge Sharing in the KDE Developer Mailing List”. In: *Management Science* 52.7 (2006), pp. 1031–1042 (*cited on pp. 10, 13*).

[21] Karim Lakhani and Eric von Hippel. “How Open Source Software Works: ”Free” User-to-User Assistance”. In: *Research Policy* 32.6 (2003), pp. 923–943 (*cited on p. 13*).

[22] Giovan Lanzara and Michèle Morner. “The Knowledge Ecology of Open-Source Software Projects”. In: *European Group of Organizational Studies Colloquium (EGOS)*. Copenhagen, 2003 (*cited on p. 13*).

[23] Greg Madey, Vincennt Freeh, and Renee Tynan. “The Open Source Software Development Phenomenon: An Analysis Based on Social Network Theory”. In: *Americas Conference on Information Systems (AMCIS)*. 2002, pp. 1806–1813 (*cited on pp. 2, 3*).

[24] “Mailing Lists | Django Documentation”. 2017-09-10. URL: `https://docs.djangoproject.com/en/dev/internals/mailing-lists/` (*cited on p. 75*).

[25] Sergio Toral Marín, Rocío Martínez-Torres, and Federico Barrero. “Analysis of Virtual Communities Supporting OSS Projects Using Social Network Analysis”. In: *Information & Software Technology* 52.3 (2010), pp. 296–303 (*cited on p. 1*).

[26] William Mendenhall, Robert Beaver, and Barbara Beaver. “Introduction to Probability and Statistics”. Thomson/Brooks/Cole, 2003 (*cited on pp. 23, 24*).

[27] Audris Mockus, Roy Fielding, and James Herbsleb. “Two Case Studies of Open Source Software Development: Apache and Mozilla”. In: *ACM Transactions on Software Engineering Methodology (TOSEM)* 11.3 (2002), pp. 309–346 (*cited on p. 13*).

[28] R Core Team. “R: A Language and Environment for Statistical Computing”. R Foundation for Statistical Computing. Vienna, Austria, 2017 (*cited on pp. 23, 31*).

[29] Michael Scialdone, Na Li, Robert Heckman, and Kevin Crowston. “Group Maintenance Behaviors of Core and Peripherial Members of Free/Libre Open Source Software Teams”. In: *International Conference on Open Source Systems (OSS)*. 2009, pp. 298–309 (*cited on pp. 4, 14, 83*).

[30] Sulayman Sowe, Ioannis Stamelos, and Lefteris Angelis. “Identifying Knowledge Brokers that Yield Software Engineering Knowledge in OSS Projects”. In: *Information & Software Technology* 48.11 (2006), pp. 1025–1033 (*cited on pp. 4, 13*).

[31] Antonio Terceiro, Luiz Romário Rios, and Christina Chavez. “An Empirical Study on the Structural Complexity Introduced by Core and Peripheral Developers in Free Software Projects”. In: *Brazilian Symposium on Software Engineering (SBES)*. 2010, pp. 21–29 (*cited on pp. 3, 4, 5, 8, 9, 12, 17*).

[32] “WineHQ - Mailing Lists/Forums”. 2017-09-10. URL: `https://www.winehq.org/forums` (*cited on p. 75*).

[33] Yunwen Ye and Kouichi Kishida. “Toward an Understanding of the Motivation of Open Source Software Developers”. In: *International Conference on Software Engineering (ICSE)*. 2003, pp. 419–429 (*cited on pp. 3, 8, 9, 11*).

Eidesstattliche Erklärung:

Hiermit versichere ich an Eides statt, dass ich diese Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, 18. September 2017

Sofie Kemper