Understanding Broadcast Based Peer Review on Open Source Software Projects

Peter C. Rigby
Software Engineering Group
University of Victoria
Victoria, BC, Canada
pcr@uvic.ca

Margaret-Anne Storey
Software Engineering Group
University of Victoria
Victoria, BC, Canada
mstorey@uvic.ca

ABSTRACT

Software peer review has proven to be a successful technique in open source software (OSS) development. In contrast to industry, where reviews are typically assigned to specific individuals, changes are broadcast to hundreds of potentially interested stakeholders. Despite concerns that reviews may be ignored, or that discussions will deadlock because too many uninformed stakeholders are involved, we find that this approach works well in practice. In this paper, we describe an empirical study to investigate the mechanisms and behaviours that developers use to find code changes they are competent to review. We also explore how stakeholders interact with one another during the review process. We manually examine hundreds of reviews across five high profile OSS projects. Our findings provide insights into the simple, community-wide techniques that developers use to effectively manage large quantities of reviews. The themes that emerge from our study are enriched and validated by interviewing long-serving core developers.

Categories and Subject Descriptors

K.6.3 [**Software Management**]: [Software development; Software maintenance; Software process]

General Terms

Experimentation, Human Factors, Management

Keywords

Peer review, Open source software, Case studies, Grounded theory

1. INTRODUCTION

Peer review by a large community of developers and other stakeholders is often championed as one of the most important aspects of open source software (OSS) development [4, 11, 14, 16], and it has been adopted by most mature, successful OSS projects [1, 15]. A review begins with an author creating a patch (a software change). This patch is broadcast to a community of potentially interested individuals. The patch can be ignored, or it can be reviewed with feedback sent to the author and also broadcast to the larger community. The author and other stakeholders revise and discuss the patch until it is ultimately accepted or rejected.

Broadcasting reviews and other development discussion to hundreds of potentially interested individuals has both significant advantages and potentially serious weaknesses. For example, we know that a diverse set of stakeholders will see the patch, but how can we be sure that any will review it and be competent enough to find defects? Additionally, there may be too many conflicting opinions during a review. How do stakeholders interact to avoid deadlock and reach an acceptable solution?

In previous work, we examined the process and quantified the parameters of review on the Apache HTTP server project [16]. In Rigby's dissertation [15], he replicated this study on the projects we examine in this paper. These and other studies (*e.g.* [1, 4, 11]) have shown OSS peer review to be both efficient and effective. However, the *underlying mechanisms and behaviours* that facilitate peer review in a broadcast setting are not well understood. In this paper, our goal is to inform researchers, developers, and managers about the advantages and weaknesses of this style of review.

This paper is structured as follows. In the next section, we introduce our research questions. In section 3, we introduce our methodology and data sources. In each subsequent section, we provide evidence regarding the techniques used to find patches for review (Q1), the impact of ignored reviews (Q2), review outcomes, stakeholders, interactions (Q3), the effect of too many opinions during a review (Q4), and scalability issues involved in broadcasting reviews in large projects (Q5). In the final sections, we discuss the threats to credibility of this study, and summarize the findings.

2. RESEARCH QUESTIONS

We explore the following research questions:

Q1 Finding patches to review: How do experts decide which contributions to review in a broadcast of general development discussion?

Previous research [1, 15] has shown that there are expert reviewers involved in most reviews on OSS projects. However, this past research does not explain how experienced developers find patches they are competent to review. We aim to understand the techniques used to wade through a seemingly overwhelming broadcast of information as well as uncover what motivates a developer to review a patch.

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA
Copyright 2011 ACM 978-1-4503-0445-0/11/05 ...$10.00.

Q2 Too few reviewers: Why are patches ignored? What is their impact on the project?

Many patches in OSS development are never reviewed [2, 11]. Why does this occur and how significant are unreviewed patches?

Q3 Stakeholders, interactions, and review outcomes: Once a patch has been selected for review, which stakeholders become involved? How do these stakeholders interact to reach a decision?

One touted strength of OSS development is the high degree of community involvement [6, 14]. How do the author, core developers, and other stakeholders, such as external developers and users, collaborate to create a working solution?

Q4 Too many opinions: What happens to the decision making process when very large numbers of stakeholders are involved in a review?

Parkinson's law of triviality, or the "bike shed painting" effect as it is known in OSS communities [6, 9], states that "organizations give disproportionate weight to trivial issues" [13]. Intuitively, a broadcast to a large community of stakeholders should exacerbate this law and lead to unimportant changes being discussed to a deadlock. We measure the impact of Parkinson's law on OSS projects and discuss the implications for broadcast based review.

Q5 Scalability: Can large projects effectively use broadcast based review?

On a large mailing list with a wide variety of topics, we expect that developers will have difficulty finding the information and patches pertinent to their work. To investigate the influence of project size on peer review, we examine five projects. We want to understand how the techniques used to manage broadcast information on a moderately sized project, like Apache, differ from a much larger project, like Linux, or a much more diverse project, like KDE.

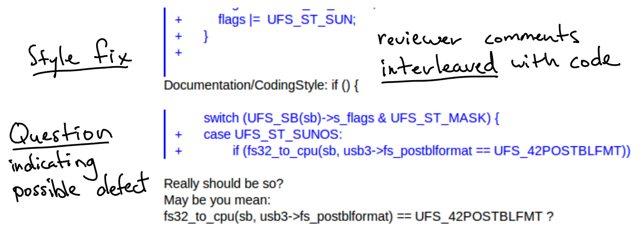
3. METHODOLOGY

Reviews occur as email threads on a mailing list [16]. An author emails a contribution of code (*i.e.* a patch) or other software artifact to the mailing list, and reviewers respond to the email with defects and other issues. These email threads, which contain reviews of patches, are our unit of analysis. They encapsulate which developers take notice of a patch and how various stakeholders interact during the review.

In this section, we describe the multiple case study methodology we used to examine the mechanisms and behaviours that underlie peer review in the context of five OSS projects. After describing the projects, we outline our use of grounded theory to code 460 instances of review and to code interview data from nine core developers.

3.1 Project Selection

We used theoretical sampling to select five projects to study [19]. We began by selecting the Apache HTTP server project because its review policies are well known and used by other projects [6]. Next, we examined the Subversion version control system as it uses a similar review process to Apache and is of similar size. Subversion was a useful first test of our preliminary themes. For the next two projects, we chose FreeBSD and the Linux kernel; both are UNIX based operating systems. These projects are significantly larger than either Apache or Subversion (see Figure 2 in Section 8). Linux also has a unique set of policies and governance structures which could yield interesting contrasts. The fifth project, KDE, is a desktop environment and represents not a single project, as was the case with the other projects, but an entire ecosystem of projects. In addition, KDE is the only project in our study which develops end user software as well as infrastructure software.


Style fix
reviewer comments interleaved with code
Question indicating possible defect

```
+          flags |=  UFS_ST_SUN;
+      }
+
Documentation/CodingStyle: if () {

       switch (UFS_SB(sb)->s_flags & UFS_ST_MASK) {
+      case UFS_ST_SUNOS:
+              if (fs32_to_cpu(sb, usb3->fs_postblformat == UFS_42POSTBLFMT))

Really should be so?
May be you mean:
fs32_to_cpu(sb, usb3->fs_postblformat) == UFS_42POSTBLFMT ?
```

Figure 1: Example fragment of review with three codes written in the margins: a type of fix, a question that indicates a possible defect, and interleaved comments.

3.2 Analysis Procedure

We randomly sampled and manually analyzed 200 email reviews for Apache, 80 for Subversion, 70 for FreeBSD, 50 for the Linux kernel, and 40 email reviews and 20 Bugzilla reviews for KDE. Saturation of the main themes occurred relatively early on, making it unnecessary to code an equivalent number of reviews for each project [7].

The analysis of these reviews followed Glaser's [7] approach to grounded theory where manual analysis uncovers emergent abstract themes. These themes are developed from descriptive codes used by the researcher to note his or her observations. Our analysis process for Apache proceeded as follows:

1. The messages in a review thread were analyzed chronologically. Since patches could often take up many screens with technical details, we first summarized each review thread. The summary uncovered high-level occurrences, such as how reviewers interacted and responded. The summaries were written without any interpretation of the events [7].
2. The reviews were coded by printing and reading the summaries and writing the codes in the margin. The codes represented the techniques used to perform a review and the types and styles of interactions among stakeholders. The example shown in Figure 1 combines the first two steps with emergent codes being underlined.
3. These codes were abstracted into memos which were sorted into themes. These themes are represented throughout the paper as paragraph and section headings.
4. A draft of the themes was given to the then president of the Apache Foundation, Justin Erenkrantz, for conceptual validation.

This process was repeated for the other four projects, with the final validation step replaced by interviews with experienced reviewers from each of the projects. While these interviews were originally intended as a validation of the coded review findings, new themes did emerge from the interviews and are an integral part of this paper. These interviews were conducted after the coding and memoing of reviews had been completed for all projects.

The semi-structured interview questions were based on the themes that emerged from manual coding of reviews on all the projects [1]. To analyze the interviews, we coded them according to grounded theory methodology.

To select interviewees for the Apache and Subversion projects, we had Erenkrantz send an email on our behalf to the respective developer mailing lists. Not knowing any developers on the remaining projects, we ranked developers based on the number of email based reviews they had performed and sent an interview request to each of the top five reviewers on each project. All these individuals were core developers with either committer rights or maintainers of a module. Initially, we asked for a phone interview; however, most developers refused to be interviewed by phone, though they were willing to answer questions by email. This was not particularly surprising since OSS developers are very experienced with and prefer communication via email. Most responses were detailed, and most developers quickly responded to follow-up questions.

Overall, we interviewed nine core developers. Some developers were willing to be quoted and are referred to by project and name, while others wished to remain anonymous and are referred to by project only. To provide a quick reference to interviewees we use a subscript of the first letter of the project and first letter of the interviewee's last name or a number in the case of an anonymous interviewee. We were able to interview one core developer from Apache in person, who preferred to remain anonymous$_{A1}$, and one Subversion developer (Wright$_{SW}$) over the phone. The remaining developers were interviewed via email: three Linux maintainers (Gleixner$_{LG}$, Iwai$_{LI}$, and Kroah-Hartman$_{LK}$) two core developers from FreeBSD who preferred to remain anonymous$_{F1,F2}$, and two core developers from KDE (Faure$_{KF}$ and Seigo$_{KS}$). Interviewees were all long-serving core developers on their respective projects.

In summary, the analysis in this paper focuses on the manual coding of 460 review discussions across five projects, as well as our interviews with nine core developers. We also use simple measures to assess the impact of certain themes. Throughout this paper, named themes are presented as section and paragraph headings.

4. FINDING PATCHES TO REVIEW

Q1: How do experts decide which contributions to review in a broadcast of general development discussion?

In contrast to an industrial setting, reviews are not assigned in OSS projects. In previous work, we have shown that although there is a potentially large number of reviewers (*i.e.* everyone subscribed to the mailing list), only a small number of reviewers (usually two reviewers) are actually involved in any given review [16]. We also showed that the typical reviewer is a long-standing committer in the area of the project to which the patch under review belongs [15].

Interviewing core developers allowed us to understand why developers decide to review a particular patch. We asked, "How do you determine what to review?" All our interviewees said that they review patches that are within their area of interest and expertise that they see as important contributions.

> "Mostly [I review because of] interest or experience in the subject area [of the patch]."
> Anonymous, FreeBSD, Interview$_{F2}$

Additionally, developers who have contributed significantly to an area of the code base often become the *defacto* owner, or in a large project, the official maintainer, assuming a degree of responsibility and a sense of obligation to perform reviews in that area$_{A1,SW,LG,LI,LK,F1,F2,KS,KF}$.

While we now have an understanding of how developers decide which patches they will review, it is surprising to us that developers are able to find these contributions in a large and general broadcast of development discussion and code changes. Some patches are assigned to or sent to specific developers, a topic we quantify and discuss in detail in Section 8.2, but for the most part, developers self-select broadcast patches for review. In this section, we present the techniques that core developers use to find and refind reviews: email filters, progressive detail within patches, an interleaved and full history of past discussions, and recipient building based on past interest.

4.1 Filtering

All interviewees agreed they receive an overwhelming quantity of broadcast information via email on a daily basis, the entirety of which they cannot process manually. Core developers reported that they subscribe to between 5 and 30 mailing lists, with most developers receiving more than 300 messages daily. This quantity of correspondence is drastically larger than the number of emails received by the average individual. For example, Whittaker and Sidner [18] found that individuals received between a mean of 40 and 60 emails per day. However, with personal email one is expected to respond to most messages, while email broadcast to a large community rarely requires a response. Instead these broadcast messages serve to keep developers aware of the general activity on the project [8].

> "If anything, there is almost too much awareness"
> Anonymous, Apache, Interview$_{A1}$

To avoid becoming overwhelmed, most interviewees separate "interesting and important" emails from general list traffic by using email filters. Since email archives do not record the filters used by developers, the following result is based exclusively on interviews. Interesting email is placed into folders that are checked on a regular basis. There are varying levels of filter sophistication, listed from simple to complex below:

- Separate tags or folders for each mailing list$_{All}$
- Differentiation between email sent directly to a developer (*e.g.* developer is on "Cc" list) and email sent generally to the list$_{LG,KF,KS}$
- Differentiation of emails that come from certain people or "important" individuals$_{LI,LK,KS}$
- Parsing the subject for interesting keywords, usually related to areas maintained by the developer$_{LG,LI,F1}$
- Parsing email for changes to files that belong to the developer's area of interest$_{LG}$

[1]The interview questions are contained in Rigby's dissertation [15]

These filtering techniques reduce information overload and allow for email to be sorted according to interest:

"Because of those filters, no, I do not feel overwhelmed [by the Linux Kernel Mailing List (LKML)]. On subsystem-specific mailing lists (like the linux-usb or linux-pci mailing lists), I read all emails"
Kroah-Hartman, Linux, Interview$_{LK}$

Within this filtered set of emails, there are still many emails that have potential relevance for a particular developer. The subject, format of the patch, and certain properties of email allow developers to identify which patches to review.

4.2 Progressive Detail

The author's email, which contains his or her patch, has three structural components that progressively provide increasingly detailed information to potential reviewers (*i.e.* progressive detail): the subject, the change log, and the file differences. Our evidence for the importance of patch structure comes from review instances where we saw authors rebuked for not following the correct format and the interviewees$_{LG,LI,LK,KF,KS}$ who said that a well structured patch made it easier to review. The following quotes are taken from our interview with Gleixner$_{LG}$.

Subject: "A good and descriptive subject line draws immediate attention."

The email subject provides an overview of the problem. All interviewees remarked on the importance of subject scanning for finding interesting patches.

Change log: "The change log message is equally important as it helps one to understand why a particular change is necessary, which problem it fixes or how the new feature is supposed to work."

The change log introduces the developer to the problem and allows them to determine, at an abstract level, whether the change is worth reviewing. While some projects require detailed change logs (*e.g.* Linux and Subversion), others do not. Our interviewees had differing opinions about the importance of the change log with some preferring to skip it and look at the changes immediately$_{F1}$.

File changes: "The diff [the code change] itself should be short, *i.e.* large patches should be split into logical and functional steps. That makes reviewing easier."

The file differences, the essential part of a patch, are the actual changes to the files.

By progressively providing increasing levels of detail, reviewers can quickly determine (1) if they have the necessary interest, skill, and time to perform a detailed review (*e.g.* if the reviewer does not understand or is uninterested in the subject, they need not look at the file changes) and (2) if the contribution is at a sufficient level of quality to warrant a detailed review.

4.3 Full and Interleaved History

When individuals perform reviews, they respond by extracting problematic sections of the review request email. Critiques or issues are written directly under the code section that has a potential defect and irrelevant text is removed (see Figure 1). This technique is known as "interleaved posting" and is widely used in email communities [2]. As the author and reviewers discuss a problem, a stack of quoted text with responses creates a focused discussion around a particular problem. Indeed, there may be multiple related and unrelated discussions in a single email thread. Seigo noted that the inability to interleave comments within code was one of the significant weaknesses of performing reviews in Bugzilla$_{KS}$.

By viewing the extracted sub-problems presented in chronological order, a reviewer can quickly form an opinion regarding an issue. The reviewer can also assess which issues have or have not been discussed, and whether the current participants are making progress towards a solution. This history is of great utility to developers on large projects where interviewees admitted they are unable to keep up with all discussions they are potentially interested in$_{LG,LI}$. In contrast, interviewees on smaller projects, such as Apache and SVN, stated they have little trouble keeping track of daily discussion and have limited need for email thread histories$_{A1,SW}$.

4.4 Refinding: Recipient Building

As we have seen, developers respond to contributions when they are interested in or feel a sense of obligation to review a patch. However, as the author revises his or her contribution, it would be very inefficient to refind this discussion thread in the breadth of broadcast information each time a change is made. The recipient fields (*i.e.* the "To" and "Cc" fields) provide a simple and natural mechanism for remaining aware of the development of a thread once interest has been indicated$_{SW,LG,LI,LK,F1,KF,KS}$. Unlike private email, where the recipients of a message remain the same unless a new recipient is explicitly added, with mailing lists, the number of recipients grows with the number of distinct people that respond. Once interest has been indicated by a response to the mailing list, subsequent responses are sent not only to the mailing list, but also specifically to the growing number of developers in the recipient fields. As we discussed above, interviewees pay more attention to emails sent to them directly than to those sent to the general list. These direct emails are responded to in a manner similar to private email.

In summary, we have learned that developers select contributions to review based on their interest in the area and in some cases, a developed obligation. We discussed the techniques that allow reviewers to locate contributions they are competent to review and to remain aware of future discussion: email filters, progressive detail within contributions, a full and interleaved history of past discussions, and recipient building based on interest in a topic. These techniques were present in all five projects examined. In the next section, we describe what happens when a contribution is neglected.

5. IGNORED PATCHES

Q2: Why are patches ignored? What is their impact on the project?

Previous work has demonstrated that a large number of patches posted to mailing lists are ignored [1, 2]. In this section, we examine what effect ignored patches have on an OSS project.

Although the traditional pressures of finishing a meeting on time do not exist in an asynchronous style of peer review, developers on OSS projects are still restricted by the amount of time they have to spend on a given project. When asked how time pressures affect the review process, interviewees stated that quality does not suffer, but that reviews of con-

[2]It is often referred to as "bottom posting" `http://en.wikipedia.org/wiki/Posting_style`

tributions are sometimes postponed or dropped$_{A1,LG,F1,KF}$. The following quotation summarizes the finding:

"Lack of time can postpone the actual reviewing, but not really have an effect on whether to review or the quality of it."

Faure, KDE, Interview$_{KF}$

While the postponement of a review, and sometimes permanent "postponement", (*i.e.* ignored low priority patches) may seem like an obvious solution, it is in stark contrast to what often happens in industry. In industry, the goal of simply reviewing everything can take priority over review quality [5]. As a deadline approaches, code may be added to the system without review and more pressing activities (*e.g.* creating a workaround so the system can ship), replace longterm activities (*e.g.* peer review) [17]. The lack of rigid deadlines in OSS development and the reduced importance of monetary work incentives appear to produce an environment where proper reviewing is seen as more important than, for example, ensuring the latest feature makes it into the product.

By postponing reviews, the burden of making sure a patch is reviewed and committed by a core developer lies with the author, not the reviewer. While this may be frustrating for the author, it means the reviewer, who is usually more experienced and pressed for time, is simply "pinged" about, or reminded by the author periodically$_{LG}$. An author's patch will only be reviewed once it is considered by a core developer to be sufficiently interesting in relation to his or her other work$_{A1,SW,LG,LI,LK,F1,F2,KF,KS}$. Inexperienced authors can be intolerant of this waiting period and may become frustrated or discouraged and provoke core developers. The following quotation is extracted from a response to a FreeBSD mailing list discussion that was started by Kamp [9]. It illustrates the sentiments of one annoyed developer:

"There is no sense in wasting the time of one informed developer to help one uninformed developer; this is a bad tradeoff unless the uninformed developer is showing signs of promise . . . Are you [the inexperienced author] willing to accept that you may have been judged 'not worth the effort' on the content of your questions [and so are ignored]?"

While textual communication removes social cues and may lead to individuals responding in what appears to be a negative or even confrontational manner, our interviewees stated that list etiquette is very important$_{LG,LI,LK}$ and that they make a concerted effort to be polite and to respond to authors$_{LG,F1}$. During our manual analysis, we found that authors who were overconfident, rude, or displayed a sense of entitlement were frequently ignored, "flamed", or reprimanded. Developers who fixed authentic problems or conducted themselves modestly usually received some form of response, even if only to point out that their contribution was redundant or outside the scope of the project. We also saw instances where novice authors ignored polite detailed reviews from core developers.

There are instances where interesting contributions can be inadvertently lost, *i.e.* ignored, in the "noise" on a mailing list. As we have seen, it is common project policy to ask that ignored contributions be reposted. Bug tracking systems can help reduce inadvertently ignored patches. However, when we examined 20 bugs on KDE's Bugzilla, we found that bug tracking may not be the solution. Although external developers' bugs were recorded and often discussed in Bugzilla, many had the complaint, "This still hasn't been committed!" For some bugs, a core developer would respond "I'll look at this tomorrow" and then never respond again. Although tracking is certainly an aspect of the ignored review problem, our findings indicate that the time, priorities, and interest of the core development team are the more central issues.

To summarize, code contributions that do not generate sufficient interest within the core development team of an OSS project are usually ignored. It is up to the author to resend his or her patch to the mailing list until a response is received. Tracking mechanisms do not appear to solve the problem of ignored reviews since reviewer time and interest are the main obstacles.

6. STAKEHOLDERS, INTERACTIONS, AND REVIEW OUTCOMES

Q3: Once a patch has been selected for review, which stakeholders become involved? How do these stakeholders interact to reach a decision?

Empirical investigations of software inspections and comparisons to the literature on group behaviour indicate that group synergy does not exist in traditional inspection meetings [17]. Extreme discipline and well defined roles inhibit teams from adapting to the strengths of individual members. The single goal of a traditional inspection meeting is to find defects; no discussion of a fix is allowed during the meeting [5]. Furthermore, time pressures can have the unintended effect of changing the goal from finding all defects to finishing the meeting on time [17].

In the previous section, we discussed how OSS developers are allowed to naturally gravitate towards contributions they are interested in or responsible for, and are competent to review. In this section, we first discuss the types of patches and review outcomes. We then describe stakeholder characteristics and ways that stakeholders with different skill sets and individuals of varying status participate in reviews.

6.1 Patch Types and Review Outcomes

Patches can be classified as follows$_{SW,LG}$:
(1) those that lead to a purely technical discussion, and
(2) those that lead to a discussion of project objectives, scope, or politics.

The different patch types lead to different styles of review and interactions among stakeholders. Technical patches are comparatively simple to review. The reviewer asks, "does the patch do what it claims to do and are there any flaws in the implementation?"$_{SW}$ Both author and reviewer accept that the patch is a necessary change to the system. In contrast, when reviewers question the value of a contribution and project scope and politics are discussed, a great deal of diplomacy can be required. Instead of focusing on technical aspects, discussions focus on whether the change is necessary and the effect it will have on other aspects of the system. In the context of the two types of patches, we list the reasons why our interviewees rejected a patch or required further modification before accepting it.

Technical issues:

- Obviously poor quality code (*e.g.* novices that are clearly out of their depth)$_{LG,KS}$

- Violation of style or coding guidelines (*e.g.* not portable code)$_{LK}$
- Gratuitous changes mixed with "true" changes (*e.g.* re-indenting a file in the same patch as a bug fix)$_{F1}$
- Code does not do or fix what it claims to or introduces new bugs$_{LK,KF,KS}$
- Fix conflicts with existing code (*i.e.* integration)$_{A1,LG}$
- Use of incorrect API or library$_{LK}$

Feature, scope, or process issues:

- Timeline prevents addition of fix or feature (*e.g.* new code during a code freeze)$_{LI}$
- Poor design or architecture$_{LG,LK}$
- Fix is misplaced or does not fit design of the system (*e.g.* optimization for a rare use case)$_{SW,KS}$
- Addition of a large codebase or feature that no individual is willing to maintain or review (*e.g.* new subsystem that is outside of the core developers' interest)$_{A1,F1}$
- Project politics (*e.g.* ideology takes precedence over objective technical discussions)$_{A1,F2}$
- Addition of code that provides no significant improvement over existing code [3]

While technical issues dominate the discussions on OSS mailing lists, project scope and other political issues can lead to tiring, vitriolic debates. However, according to our interviewees, this level of conflict is rare (see Section 7) as core developers have a preference to keep what works over adding a new feature with questionable value.

> "I'm not claiming this [proposal ...] is really any better/worse than the current behaviour from a theoretical standpoint, but at least the current behaviour is _tested_, which makes it better in practice. So if we want to change this, I think we want to change it to something that is _obviously_ better."
> Torvalds, `http://kerneltrap.org/mailarchive/linux-kernel/2007/7/26/122293`

6.2 Stakeholder Characteristics

When asked, "Which roles do you see during a review?", interviewees seemed confused by our question and, instead of roles, described characteristics or manners in which stakeholders interacted.

In our manual analysis of reviews, two main general roles emerged: reviewer and outsider. Interviewees added subtle depth further describing the characteristics and interactions during a review.

6.2.1 Reviewer Characteristics

Our analysis of reviews revealed several reviewer characteristics. These were supported by discussion with our interviewees. The terms used by Gleixner$_{LG}$ of the Linux project were particularly descriptive and we have used them below: objective analyzer, fatherly adviser, grumpy cynic, frustratedly resigned, and enthusiastic supporter. We divide them into positive and negative personas.

[3]This theme was found only during manual analysis of reviews

Positive Personas

Objective analyzer: All reviewers are expected to provide a disinterested analysis of the patch. This style of analysis is the goal and intended purpose of peer review. Interestingly, reviewers often provide their criticisms as questions, which can forestall adversarial behaviour and encourage objective discussion, use cases, and "what if" scenarios (see Figure 1).

Expert or fatherly adviser: New developers may not understand all the subtleties and quirks of both the system and the development community$_{KF}$. This experience is built up over time. Expert reviewers often provide *ad hoc* mentoring$_{SW,LG}$ to novices by, for example, pointing out dead ends, providing links to email discussions of related past issues, and passing on the culture of the community. Sometimes the "fatherly adviser" will simply rewrite the code himself instead of fixing the novice's convoluted patch$_{LG,F2,KS}$.

Enthusiastic supporter or champion: When the author of a patch is not a committer (*i.e.* not a core developer), an enthusiastic reviewer or champion must shepherd the patch and take responsibility for committing it. If no one champions an outsider's patch, it will not be committed.

Champions are also present when it is unclear what the best solution is. In this case, different developers put their support behind different competing solutions, indicating the feeling of various segments of the community.

Negative Personas

Grumpy cynic: Experienced project members can become cynical when, for example, novices repeatedly suggest a "solution" that has been explored in the past with an outcome of failure. "I usually try to be calm and polite, but there are times where sarcasm or an outburst of fury becomes the last resort."$_{LG}$

Frustratedly resigned: When a review discussion has been carrying on for an extended period of time, a reviewer may simply resign from the discussion and refrain from commenting. Since "silence implies consent" in OSS communities [16], the developers who continue the discussion ultimately make the decisions that influence the system. When any reasonable solution will suffice, the most effective approach is to leave the discussion. These fruitless discussions are discussed in detail in Section 7.

6.2.2 Outsiders

Having a large community of knowledgeable individuals, outside of the core development team, who can provide feedback and help with testing, is one of the strengths of OSS development [14].

However, unlike a core developer who has been vetted and voted into the inner circle or "appointed" as a maintainer of a subsystem, outsiders come from a wide swath of experience and expertise, which can vary drastically from that of the core development team. These differences can be both an asset and an obstacle. The interviewed core developers had mixed feelings about the contributions from outsiders. Outsiders made positive contributions by suggesting real world use cases and features$_{A1,SW,LG}$, providing expert knowledge that the core team lacks$_{A1,F1}$, testing the system and the APIs$_{SW}$, providing user interface and user experience feedback$_{F1}$, and filing bug reports$_{LG,F2}$ and preliminary reviews$_{KF,KS}$. According to interviewees, negative contributions came in the form of, often rude, requests for features that had already been rejected for technical reasons$_{LG}$, as well as use cases and code optimizations that were too specific to be relevant to the general community$_{SW}$.

The themes of competence and objectivity emerged from the interviews and manual analysis as attributes that make an outsider an important contributor to OSS projects$_{SW,LG,F1,F2}$. Competence need not be technical when it comes from an outsider, but it is usually in an area that is lacking in the core development team. In general, non-objective opinions are shunned in this meritocratic community.

While most will be of limited help$_{LI,F1,F2,KF}$, there are a small number of outsiders that are critical to the success of an OSS project$_{SW,LK,F1}$. For example, external developers who typically write code for a subproject that depends on the software being developed by the main project make critical contributions$_{SW,F1}$. In the case of the Apache foundation, the httpd server is used by other modules that have a larger and faster growing code base than the httpd server itself. Developers working on these dependent modules find bugs and provide use cases that are based on objective, real world experience. They also often provide competent code and reviews during the resolution of a bug that affects their project.

In summary, the shift away from explicit roles and time constrained meetings has allowed peer review to move from a purely defect finding activity to a group problem solving activity that recognizes the varying strengths of participants. Patches can lead to interactions between stakeholders that are purely technical or that involve scope and political issues. Reviewers take on various personas and competent and objective outsiders contribute knowledge lacking in the core team. Outsiders can also hinder the review process, a topic we quantify in the next section.

7. BIKE SHED DISCUSSION: TOO MANY OPINIONS

Q4: What happens to the decision making processes when very large numbers of stakeholders are involved in a review?

A disadvantage of broadcasting to a large group is that too many people may respond to a patch, slowing down the decision process. In traditional organizations, this problem is known as Parkinson's Law of Triviality [13]. It has been repopularized in the OSS community by Kamp as the "painting the bike shed" effect [9] – it does not matter what colour a bike shed is painted but everyone involved in its construction will have an opinion.

Our interviewees were aware of the bike shed effect and many referred to it explicitly$_{A1,SW,LG,F2}$. They dreaded these types of discussions and made every effort to avoid them. Some projects have explicit policies on how to avoid the bike shed effect. For example, on the SVN project, when a core developer says "this discussion is bike shedding," core developers are supposed to ignore any further conversation on the thread [6]. Interviewees stated that this level of conflict was rare$_{A1,SW,F1}$ and usually involved uninformed outsiders$_{LG,LI,F2}$. However, we wished to quantify how influential outsider opinions are on OSS projects and how often bike shedding does occur.

To create a measure of the bike shed effect, we first note that in Fogel's experience, "bike shed" problems are identifiable by a lack of core group members in a large discussion thread [6]. We divide all stakeholders involved in all reviews into two categories: core developers, who have commit privileges, and outsiders, who do not [4].

	Apache	SVN	FreeBSD	KDE
Percentage of reviews:				
Outsiders involved	30%	23%	13%	18%
Outsiders majority	5%	3%	2%	4%
"Outsiders' influence" measure correlated with:				
Time for review	−.17	−.24	−.32	−.20
Total messages	−.53	−.59	−.60	−.38
Core-dev messages	−.76	−.77	−.77	−.63

Table 1: The influence of outsiders during the review process

If the bike shed effect is a significant problem on OSS projects, we would expect to see many reviews dominated by outsiders. Table 1 shows that the number of messages sent by outsiders only exceeded that of the core development group in between 2% (in the case of FreeBSD), and 5% (in the case of Apache) of the time. However, outsiders were involved in between 13% (for FreeBSD), and 30% (for Apache) of all review discussions.

We also create an "outsiders' influence" metric by dividing the number of outsider messages by the total number of messages in the review thread. In examining threads that had at least one outside reviewer, we found that outsider influence was weakly correlated with threads that lasted for a *shorter* period of time ($-0.17 \leq r \leq -0.32$), moderately correlated with *fewer* total messages ($-0.38 \leq r \leq -0.60$) and strongly correlated with *fewer* messages from core developers ($-0.63 \leq r \leq -0.77$) [5]. These negative correlations indicate that outsiders receive less attention than core developers.

While outsiders are clearly involved in a significant number of reviews, the "bike shed" effect, which Fogel defines as a long discussion that involves few core developers, does not appear to be a significant problem during OSS reviews. Our findings appear to support Bird *et al.*'s [3] finding that individuals with lower status in an OSS community (*i.e.* outsiders) generally receive little attention from the community.

8. SCALABILITY

Q5: Can large projects effectively use broadcast based review?

Intuitively, broadcasting information to a group of individuals should become less effective as the number and range of discussions increase. Our goal is to understand if and how large projects, Linux, FreeBSD, and KDE (see Figure 2), have adapted the broadcast mechanism to maintain awareness and peripheral support of others, while not becoming overwhelmed by the number of messages and commits. The themes of multiple topic-specific mailing lists and explicit review requests to developers emerged on the periphery of our previous analysis as potential effective scaling techniques. In this section, we focus on these themes and develop research questions that allow us to understand the advantages and disadvantages of using each technique to manage the review process on large projects.

[4]We were unable to use this measure for Linux because there is no central version control repository and commit privileges are not assigned

[5]All correlations are statistically significant at $p < 0.01$

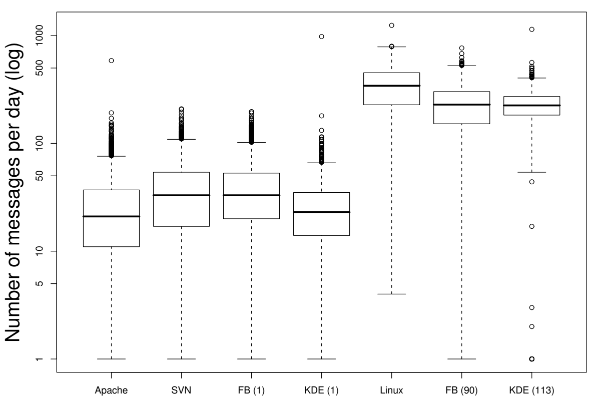


Figure 2: Scale of the project by emails sent daily

8.1 Multiple Lists

Large projects have more than one mailing list that contains review discussions. Lists have specific charters that isolate developers working on a specific topic, or set of related topics, into a workgroup$_{A1,SW,LG,LI,LK,KF,KS}$. These divisions also allow developers to create email filters based on the mailing lists in which they are interested, as discussed in Section 4.1.

Do multiple mailing lists reduce the amount of traffic on individual lists?
Figure 2 shows that FreeBSD has 90 mailing lists with an overall median of 233 messages per day, while its busiest single mailing list has a median of 34. KDE has 113 mailing lists with an overall median of 226 messages per day, while its busiest single mailing list has a median of 20. The development lists for Apache and SVN have a median of 21 and 34 messages, respectively. Some of our interviewees discussed how they were able to keep up with all the messages on topic specific lists$_{A1,SW,LI,LK,F2}$ (*e.g.* quotation in Section 4.1). With the exception of the Linux Kernel Mailing List [6], which has a median of 343 messages per day and is discussed in the next section, it appears that while the total amount of traffic on large projects is unmanageable for any individual, an individual should be able to follow all the traffic on a specific list.

Do topic specific lists become isolated and lose valuable outsider input?
One worry with list specialization is that developers who are peripherally related to a discussion will not be made aware of it since they are not subscribed to that particular list. In this way, valuable outsider input could be lost. Large projects dealt with this issue by having individuals who bridge multiple lists.

There are two types of bridging: discussion threads that span more than one list and individuals who are subscribed and discuss topics on more than one list. Our interviewees stated that they subscribe to between 5 and 30 mailing lists (see Section 4) and that while cross-posting does happen, it is generally rare and only occurs when a issue is felt to be related to an aspect of the system covered by another mailing list$_{SW,LG,LI,LK,F1,F2,KF,KS}$. Discussions can also be moved from more general lists to specific lists.

How often do review threads span multiple lists? How many lists do individuals comment on?
While our interviewees indicate that mailing lists are bridged, we use three measures to quantify the strength of relationships between lists. In the context of review threads, we examine how many messages are posted to more than one list, how many threads span more than one list, and how many lists an individual has commented on.

We find that few messages are posted to more than one list and few review discussions span more than one list: 5% and 8% respectively for FreeBSD, and 1% and 15% respectively for KDE. That cross-posting only occurs between 5% and 15% of the time is not surprising, as the lists are set up to refer to modular components of the system. High levels of cross-posting might indicate that lists need to be merged.

In contrast to the relatively small proportion of cross-posted threads, many individuals posted to multiple lists – 30% and 28% of all individuals [7] have posted to at least two lists for FreeBSD and KDE, respectively. Only 8% and 5% of individuals posted to more than three lists for FreeBSD and KDE, respectively. However, on the FreeBSD project, one individual posted to 24 different lists.

In summary, multiple topic-specific mailing lists appear to be one technique used by large projects to deal with what would otherwise be an overwhelming amount of information. These lists allow developers to discuss details that may not be relevant to the larger community and then to sometimes post final changes to more general lists for further feedback. The mailing lists for FreeBSD and KDE appear to be relatively well separated as there are few cross-posted messages and threads. Approximately one third of the community is involved in reviews on more than one list. These individuals bridge multiple lists, allowing for outsider input on otherwise isolated lists.

8.2 Explicit Requests

In traditional inspection, the author or manager explicitly selects individuals to be involved in a review (*e.g.* Fagan [5]). With broadcast based review, the author addresses a patch email to the mailing list instead of particular individuals; thus, information an author has about potential reviewers is unused. Explicit requests or messages sent directly to potential reviewers as well as the entire mailing list can combine an author's knowledge with the self-selection inherent in broadcasting. Since core developers filter messages and pay more attention to messages sent directly to them (see Section 4), if an author wants to increase the chance that a particular reviewer will comment on a patch, the author could explicitly include the reviewer in the "To" or "Cc" fields of an email. In this section, we measure the impact of the author explicitly addressing reviewers in the Linux project.

On the Linux project, 90% of patch review requests had more than one recipient (*i.e.* the mailing list and one or more individuals). Compared to the following projects, this is remarkably high: Apache 9%, SVN 12%, FreeBSD 27%, and KDE 13%. Since there are relatively few explicitly addressed emails, with the exception of Linux, it appears that most of the projects examined rely on multiple, small mailing lists

[6]Although Linux does many separate mailing lists, in this work we examine only the main development list: the LKML.

[7]In this case an "individual" is represented by a distinct email address

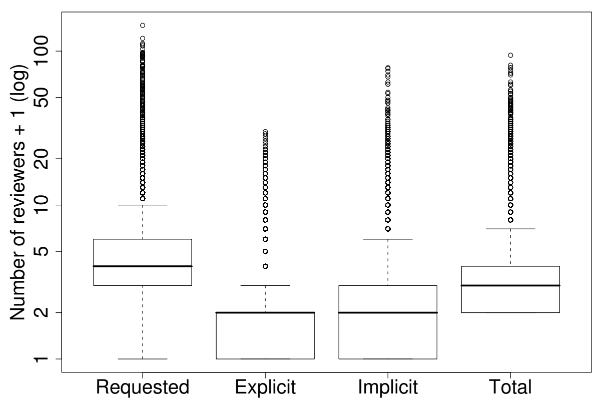


Figure 3: Explicit vs. Implicit Responses

to ensure that interested reviewers find patches they are competent to review. Given that the Linux mailing list has a median of 343 messages per day, more than the combined multiple lists on FreeBSD or KDE, and that there is a much higher percentage of explicit review requests, we decided to investigate the following:

Does the Linux list represent a breakdown in broadcast mechanism whereby self-selection is ineffective and review requests must be explicitly sent to expert reviewers?

To investigate this phenomenon, we measured the following for each review thread on Linux: (1) the number of people who were explicitly addressed – explicit request (or requested), (2) the number of people who responded that were explicitly addressed – explicit response, (3) the number of people who responded that were *not* explicitly addressed – implicit response, and (4) the total number of people who responded – total response. Since individuals may use multiple email addresses, we resolved all email addresses to single individuals as per Bird *et al.* [3]. Furthermore, a mailing list can only receive email, so we used regular expressions and manual examination to determine which addresses were mailing lists and which were individuals. Explicit requests sent to other mailing lists were removed from this analysis, but could be examined in future work.

If there is indeed a breakdown in the broadcast mechanism, we would expect to see few implicit responses. Figure 3 shows that this is not the case. The median number of people explicitly requested is three. The median number of people who explicitly responded is one and the median number of people who responded without an explicit request is one, giving a total of two reviewers. A Wilcoxon test failed to show a statistically significant difference between the number of people who explicitly responded compared to the number that implicitly responded. Also, a Spearman correlation test revealed that the number of explicitly requested individuals had almost no effect on the number of individuals that implicitly responded, $r = -.05$. However, a moderate correlation existed between the number of explicitly requested individuals and the total number of respondents, $r = .30$ [8].

[8]All Spearman correlations are statistically significant at $p < 0.01$

These results show that, for the Linux project, although the original message usually has explicit recipients, this does not stop other developers from responding, and implicit responses occur at least as often as explicit responses. Despite high levels of explicit requests to potentially expert reviewers, Linux sees a nearly equal number of implicit responses (*i.e.* self-selection). As such, Linux appears to maintain the advantages of broadcasting reviews, while allowing authors to notify reviewers they believe should examine their contribution.

9. THREATS TO CREDIBILITY

In this section, we discuss the internal and external credibility of our research [12]. Internal credibility assesses how meaningful and dependable our findings are within the phenomenon and context under study. While external credibility assesses the degree to which our findings generalize across different populations and contexts.

Internal Credibility. While there are many aspects to internal credibility the goal is to examine the "credibility of interpretations and conclusions within the underlying setting or group." [12] We assess the fit and level of triangulation of our findings.

Fit: Do the findings resonate with the interviewed core developers? A summary of the research findings was sent to all participants. We also asked them to search for their interviewee code to ensure that statements they had made were not misinterpreted or taken out of context. Of the nine interviewees, A1, LG, LK, F1, and KF responded. The interviewees were interested in and agreed with the findings, with LG and LK pointing out two minor issues (a missing footnote and a typo).

Triangulation "involves the use of multiple and different methods, investigators, sources, and theories to obtain corroborating evidence" [12]. We used multiple data sets (*i.e.* archival and interview data), and multiple methods (*i.e.* grounded theory and various measures of review) to triangulate our findings.

External Credibility. "External credibility refers to the degree that the findings of a study can be generalized across different populations of persons, settings, contexts, and times." [12]

We examined publicly available data on five successful OSS projects, so it should be possible to replicate our study on these or other projects. To improve the generalizability of our results, the case studies were chosen in a sequence that tested weaker aspects of our themes [19]. However, we examined only large, mature projects, so future work is required to examine the review processes of smaller and less successful projects.

Furthermore, KDE is the only project that included non-infrastructure software. KDE uses tracker based review, while post-commit review was still conducted on the mailing list. Studies of peer review on projects developing end-user software, *e.g.* Firefox and Eclipse [4, 11], indicated that when defects are reported by non-developers, improved defect and review tracking tools are required. We only interviewed core developers; while these developers do the majority of the work on an OSS project [10], attending to the needs of and interviewing outside developers is left to future work.

10. SUMMARY AND CONCLUSION

In this paper, we conducted five case studies, manually coding hundreds of reviews, interviewing core developers, and measuring several aspects of OSS review processes. Our goal was to understand the mechanisms and behaviours that facilitate peer review in an open source setting. We summarize our results below.

(1) How do experts decide to review contributions in a broadcast of general development discussion?
Developers use filtering, progressive detail within patches, full and interleaved histories of discussion, and recipient building based on past interest to find contributions that fall within their area of interest and obligation.

(2) Why are patches ignored? What is their impact on the project?
Developers prefer to postpone reviews rather than rush through them. Therefore, patches that fail to generate interest among the core development team tend to be ignored until they become interesting. It follows that issue tracking does not completely solve the problem of ignored patches.

(3) Once a patch has been selected for review, which stakeholders become involved? How do these stakeholders interact to reach a decision?
Roles are not assigned in OSS development. This allows competent and objective stakeholders to play a role that accords with their skillset and leads to group problem solving. The style of interaction among stakeholders depends on whether the patch can be assessed from a technical perspective or whether project scope must also be considered.

(4) What happens to the decision making process when very large numbers of stakeholders are involved in a review?
The latter type of patch can lead to politicized, long, opinionated and unproductive discussion, *i.e.* bike shed painting discussion. These discussions are relatively rare, and core developers typically try to avoid them.

(5) Can large projects effectively use broadcast based review?
While investigating scale issues, we found that two large projects, FreeBSD and KDE, effectively used multiple topic specific lists to separate discussions. Developers are typically subscribed to multiple lists, which results in lists being bridged when a discussion crosscuts topics. The Linux Kernel mailing list is an order of magnitude larger than the topic specific lists on the projects we examined. On this list, explicit requests to reviewers who the author believes should see a patch are employed, alongside the general broadcast.

OSS development continues to surprise software engineering researchers with its unique and often unintuitive approaches to software development that nevertheless result in mature and successful software products [10, 8]. Organic, developer driven, community-wide mechanisms and behaviours that use simple tools that rely heavily on the human can effectively manage large quantities of broadcast reviews and other development discussions.

11. ACKNOWLEDGEMENTS

We would like to thank the interviewees for participating in this study. We also thank Daniel German, Gargi Bougie, and the Chisel Group for their feedback and support.

12. REFERENCES

[1] J. Asundi and R. Jayant. Patch Review Processes in Open Source Software Development Communities: A Comparative Case Study. *HICSS*, 0:166c, 2007.

[2] C. Bird, A. Gourley, and P. Devanbu. Detecting Patch Submission and Acceptance in OSS Projects. *In MSR '07*, 2007.

[3] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. pages 137–143, 2006.

[4] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *CSCW '10*, pages 301–310, 2010.

[5] M. Fagan. *A history of software inspections*. 2002.

[6] K. Fogel. *Producing Open Source Software*. O'Reilly, 2005.

[7] B. Glaser. *Doing grounded theory: Issues and discussions*. Sociology Press Mill Valley, CA, 1998.

[8] C. Gutwin, R. Penner, and K. Schneider. Group awareness in distributed software development. In *CSCW '04*, pages 72–81, 2004.

[9] P.-H. Kamp. A bike shed (any colour will do) on greener grass... FreeBSD mailing list archive `http://www.webcitation.org/5ZZaDOxyW`, 1999.

[10] A. Mockus, R. T. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[11] M. Nurolahzade, S. M. Nasehi, S. H. Khandkar, and S. Rawal. The role of patch review in software evolution: an analysis of the mozilla firefox. In *IWPSE-Evol '09*, pages 9–18, 2009.

[12] A. Onwuegbuzie and N. Leech. Validity and Qualitative Research: An Oxymoron? *Quality and quantity*, 41(2):233–249, 2007.

[13] C. N. Parkinson. *Parkinson's Law: The Pursuit of Progress*. John Murray, 1958.

[14] E. S. Raymond. *The Cathedral and the Bazaar*. O'Reilly and Associates, 1999.

[15] P. C. Rigby. Understanding Open Source Software Peer Review: Review Processes, Parameters and Statistical Models, and Underlying Behaviours and Mechanisms. `thechiselgroup.org/rigby-dissertation.pdf`, Dissertation, 2011.

[16] P. C. Rigby, D. M. German, and M.-A. Storey. Open Source Software Peer Review Practices: A Case Study of the Apache Server. In *ICSE '08*, pages 541–550, 2008.

[17] C. Sauer, D. R. Jeffery, L. Land, and P. Yetton. The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research. *IEEE Trans. Softw. Eng.*, 26(1):1–14, 2000.

[18] S. Whittaker and C. Sidner. Email overload: exploring personal information management of email. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 276–283, 1996.

[19] R. K. Yin. *Case Study Research: Design and Methods*, volume 5 of *Applied Social Research Methods Series*. Sage Publications Inc., 2 edition, 1994.