Open Source Software Peer Review Practices: A Case Study of the Apache Server

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

Daniel M. German
Software Engineering Group
University of Victoria
Victoria, BC, Canada
dmg@uvic.ca

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

ABSTRACT

Peer review is seen as an important quality assurance mechanism in both industrial development and the open source software (OSS) community. The techniques for performing inspections have been well studied in industry; in OSS development, peer reviews are less well understood. We examine the two peer review techniques used by the successful, mature Apache server project: review-then-commit and commit-then-review. Using archival records of email discussion and version control repositories, we construct a series of metrics that produces measures similar to those used in traditional inspection experiments. Specifically, we measure the frequency of review, the level of participation in reviews, the size of the artifact under review, the calendar time to perform a review, and the number of reviews that find defects. We provide a comparison of the two Apache review techniques as well as a comparison of Apache review to inspection in an industrial project. We conclude that Apache reviews can be described as (1) early, frequent reviews (2) of small, independent, complete contributions (3) conducted asynchronously by a potentially large, but actually small, group of self-selected experts (4) leading to an efficient and effective peer review technique.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: [Metrics]; K.6.3 [**Software Management**]: [Software development; Software maintenance; Software process]

General Terms

Management, Measurement

Keywords

Peer review, Inspection, Open source software, Mining software repositories (email)

ICSE'08, May 10–18, 2008, Leipzig, Germany.

1. INTRODUCTION

Over the years, inspections (formal peer reviews) have been perceived as a valuable method to improve the quality of a software project. Inspections typically require periodic group reviews. Developers are expected to prepare for the meeting by studying the artifact under review, and then they gather to discuss it [6].

In practice, adoption remains low as developers and organizations complain of the time commitment required for inspection (and the corresponding cost) and the difficulty involved in scheduling them [10]. This problem is compounded by tight schedules in which it is easy to ignore peer reviews.

Given the difficulties with adoption of inspection techniques in industry, it is surprising that a group of mostly volunteer developers, such as those in the open source software (OSS) Apache HTTP server project (the focus of this paper), have embraced code review as one of their most important quality control techniques [8]. These developers have produced a mature, successful [16] product with defect densities similar to those found in industry [14, 22].

Very little is known in the scientific literature about how peer reviews are conducted in OSS. There are experience reports [13, 21], descriptions at the policy level [4, 14], and empirical studies that assess the level of participation in peer reviews [1, 12]. However, many of the questions traditionally asked about inspection techniques have not been answered for OSS.

Our goal is to determine empirically the parameters of Apache peer review and to compare them to those found in traditional inspections. By understanding these parameters, we expand what is known about OSS and can identify techniques from Apache peer review that might be valuable and transferable to industry. We have chosen to examine a single project because space limitations would not allow us to provide sufficient depth and detail with multiple projects.

The organization of this paper is modeled on Mockus et al [14] and has the following sections. Section 2 discusses the background and policy of the Apache peer review techniques and introduces our research questions. Section 3 describes our methodology and data extraction techniques. Section 4 answers our research questions using archival data and provides a comparison of the two Apache review techniques based on efficiency and effectiveness. Section 5 provides a comparison of Apache review to an industrial inspection process in terms of efficiency. In sections 6 and 7 we summarize our results, propose a preliminary theory of OSS peer review, and discuss necessary case study replications. The final section discusses contributions and conclusions.

2. BACKGROUND AND POLICY

The Apache server project was started by a group of globally distributed server administrators who volunteered to share fixes to the original code base of the NCSA server [8]. Since the original developers were all volunteers who had never met in a face-to-face manner, it would seem natural that they would examine each other's code before including it in their own local server. As the project evolved, a shared version control repository was created and a trusted group of developers controlled access to this repository [8]. In the Apache project, trusted developers, or core-group members, are developers who have shown their competence and have been given voting and commit privileges. The review technique, known as *review-then-commit* (RTC), was formalized and accepted as Apache policy[1]: before any code can be added to the shared repository (i.e., committed) it must receive a positive review by at least three core-group members. Interestingly, a similar phenomenon sometimes occurs when a new developer joins an industrial project. Any code written by this new developer must be reviewed by his or her mentor before it can be released to other developers on the project.

Reviewing all code before it is committed can become a very onerous, time consuming process. This frustration is illustrated through an email quotation from one of the core developers of the Apache project in 1998[2].

> `This is a beta, and I'm tired of posting bloody one-line bug fix [contributions] and waiting for [reviews] ... Making a change to apache, regardless of what the change is, is a very heavyweight affair.`

After a long, heated email discussion, a new style of review was accepted as Apache policy, namely *commit-then-review* (CTR). CTR is used when a trusted developer feels confident in what he or she is committing. RTC is now only used when core-group members are less certain of their contribution (e.g., it is complex) or when contributions are made by non-core developers (i.e., developers without commit privileges). According to Fielding, one of the founding members of the Apache project, Apache developers are expected to review all commits and report issues to the mailing list [14]. However, other core developers involved in the policy debate in 1998 are less convinced that this post-commit review will happen in a volunteer environment; one developer dubs the policy "commit-then-whatever".

A third style of review, used only by core-group members, is to develop a "lazy consensus" around their contribution. With this type of review, a contribution is submitted for review, but includes a date (e.g., two days) at which time the developer will commit the contribution provided there are no complaints. This review type is based on "silence implying consent".

Regardless of the review type used, there are two important policies when conducting a review. First, a contribution must be small, independent, and complete. Reviewers do not want to review half-finished contributions or contributions that involve solutions to unrelated problems. Large contributions can take longer to review, which can be problematic for volunteer developers. Second, if any core-group member feels that a contribution is unacceptable, he or she can place a veto on it and the contribution will either not be committed or, in the case of CTR, removed from the shared repository.

In summary, Apache's review policies create two barriers to "buggy" code. The first barrier is to require significant changes and changes from non-core developers to be reviewed before being committed (RTC). The second barrier is to require reviews of committed contributions (CTR). For the latter process to be effective, developers must examine the commit mailing list to ensure that unacceptable contributions are fixed or removed.

2.1 Research Questions

We base our research questions upon those that have been asked and answered in the past for inspection techniques (e.g., Porter et al. [18]). Where necessary, adaptations have been made to the questions to deal with the available data and aspects that are unique to the Apache project and its polices. In this section, we provide the rationale behind each question and what, if anything, is known about OSS peer review.

Q1. Frequency and Activity: Are developers able to maintain an adequate level of review during times of increased development activity?

Apache review policies enforce a review around the time of commit. As development activity increases, as there are more contributions and commits, are developers able to maintain an adequate level of peer review? This concern is especially relevant in the case of CTR where an ignored commit becomes part of the product without ever being reviewed (i.e., "commit-then-whatever"). To address this concern, we correlate review frequency to development activity.

Q2. Participation: How many reviewers respond to a review? How much discussion is there during a review? What is the size of the review group?

In his experience-based analysis of the OSS project Linux, Raymond coined Linus's Law as "Given enough eyeballs, all bugs are shallow" [21]. It is important to gauge participation in the peer reviews to assess the validity of this statement. RTC policy states that three reviewers should be involved in each review, while CTR contributions are supposed to be reviewed by the core-group. Research into the optimal number of inspectors has indicated that two reviewers perform as well as a larger group [18, 24]. Previous OSS research has found that there are on average 2.35 reviewers who respond per review for Linux [12]. We have have found a similar result for Apache [23], as have Asundi and Jayat [1]. One problem with the previous metrics is that reviewers who do not respond (i.e., they may find no defects) will not be counted as having performed a review. We measure the size of the review group at monthly intervals. Our assumption is that if a developer is performing reviews, he or she will eventually find a defect and respond.

Q3. Size: Why is the size of the artifact under review so small?

Mockus et al. [14] found that the size of a change for the Apache project was smaller than for the proprietary projects they studied, but they did not understand why. We provide a discussion relating Apache policy and practice to the size of the review and compare Apache to an industrial project based on the size of the contribution.

[1]The chief source of information pertaining to Apache policy is the Apache website `apache.org` March 2007

[2]All quotations and references to email discussion come from manual examination of the Apache developer mailing list.

We want to understand whether the small change size is a necessary condition for performing an Apache style of review.

Q4. Review Interval: What is the calendar time to perform a review?
The review interval, or the calendar time to perform a review, is an important measure of review effectiveness [18]. The speed of feedback provided to the author of a contribution is dependent on the length of the review interval. Interval has also been found to be related to the timeliness of the project. For example, Votta [11] has shown that 20% of the interval in a traditional inspection is wasted due to scheduling. We create a measure of interval and compare Apache to an industrial project.

Q5. Defects: How many reviews find defects?
The number of defects found in a review is a common but limited measure of review effectiveness [7, 10, 18]. There have been estimates of the number of defects found and fixed in OSS projects (e.g., [4, 5]). Using automated checkers, Reasoning Inc. [22] concluded that the Apache project has a defect density comparable to proprietary software. Mockus et al. [14] found that this was accomplished without a policy requiring substantial code reviews before a release. They comment that this result "may indicate that fewer defects are injected into the code, or that other defect-finding activities such as inspections are conducted more frequently or more effectively". The measure of defects in this case was the number of defects reported in the Apache bug database. Neither study determined how many of these defects are discovered through peer review. We measure the proportion of reviews that find defects.

3. METHODOLOGY AND DATA SOURCES

Apache developers rarely meet in a synchronous manner, so almost all project communication is recorded [8]. The Apache community fosters a public style of discussion, where anyone subscribed to the mailing list can comment. Discussions are usually conducted on a mailing list as an email *thread*. A thread begins with an email that includes, for example, a question, a new policy, or a contribution. As individuals reply, the thread becomes a discussion about a particular topic. If the original message is a contribution, then the discussion is a review of that contribution. We examine the threaded reviews on the developer and commit mailing lists. We also examine commit logs to determine who performed a review. One advantage of this archival data is that it is publicly available, so our results can be easily replicated.

The most important forum for development-related discussion is the developers' mailing list. All contributions that require discussion must be sent to this list. There were 84,784 email messages and 23,409 threaded discussions on the developer mailing list between January 1997 and October 2005. We examine discussions pertaining to peer review.

RTC. From 1997 on, all contributions that are reviewed before being committed (RTC) have an email subject containing the keyword "[PATCH]". This original message becomes the contribution, and all replies to this message become reviews and discussion pertaining to the contribution. There were 2,603 contributions and 9,216 replies to these contributions. We eliminated contributions in which there was no response or only the author responded, as these contributions are likely not reviewed. This approach eliminates 583 contributions, reducing the number of RTC contributions to 2,020.

CTR. Since the version control system automatically begins each commit email subject with "cvs [or svn] commit:", all replies that contain this subject are reviews of a commit. In this case, the original message in the review thread is a commit recorded in the version control mailing list; all responses still go to the developer mailing list. There were 2,647 commits that received at least one reply and 9,833 replies to these commits. We eliminated 210 commits because only the author replied; thereby reducing the number of commits we examined to 2,437. Although CTR was used prior to 1997, it did not become an official Apache policy until 1998.

Limitations of the data. The data can be divided into two sets: contributions that receive a response and contributions that do not. In this paper, we limit our examination to contributions that receive a response, because when a response occurs, we can be sure that an individual took interest in the contribution. If there is no response, we cannot be certain whether the contribution was ignored or whether it received a positive review (i.e., no defects were found). Our measurements would be skewed by ignored contributions, so we cannot include these data. For example, since an ignored contribution has no reviewers, if we include these data, we drastically reduce the number of reviewers per contribution, even though these contributions do not actually constitute reviews. Furthermore, we assume that contributions that received a positive review will use the same or fewer resources as contributions that receive a negative review. For example, we expect the review interval to be shorter when no defect is found than when one is found. In summary, we are forced to use a sample of Apache reviews (the sample is not random). We suggest that our sample is the important and interesting section of the data (i.e., it is the data that received a response). We also suggest that using "reviews" that do not receive a response would significantly reduce the usefulness of our data.

Within the set of reviews that received a response (i.e., the data we have sampled), we make an additional assumption that *a reply to a contribution is a review*. We selected a random sample of these data: 100 RTC and 100 CTR reviews. This random sampling indicated that less than 5% of the contributions contained discussions unrelated to review (e.g., policy discussions or indications that the contribution was not interesting and would not be reviewed). Thus, we feel that our assumption is reasonable.

We recognize that at least two important questions cannot be answered using the given data. First, since we cannot differentiate between ignored and positively reviewed contributions, we cannot address the exact proportion of contributions that are reviewed. Second, although we do provide the calendar time to perform a review (interval), we cannot address the amount of time it takes an individual to perform the review. However, in Lussier's [13] experience with the OSS WINE project, he finds that it typically takes 15 minutes to perform a review, with a rare maximum of one hour.

Additional data sources. There are two additional data sources. The first, manual examination, involves reading the review thread. These data provide useful qualitative evidence in the form of quotations as well as an opportunity to categorize the contributions, such as when random sam-

pling is used to determine how many contributions contained defects.

The second type involves looking at the commit log to determine who was involved in an RTC review. Apache policy requires developers to describe the changes made in each transaction to the version control system. This description, the commit log, includes information such as who submitted and who reviewed the contribution. According to Fielding [14], developers create a complete commit log more than 90% of the time. There were 2,373 commits that had at least one reviewer named in the commit log. Since all these contributions have been reviewed-then-committed and accepted, we refer to them as RTCA. The main limitation of these data is that we do not have the associated review thread, thereby limiting the metrics we can calculate with RTCA data.

Extraction Tools and Techniques. We created scripts to extract the mailing list and version control data into a database. An email script extracted the mail headers including sender, in-reply-to, and date headers. The date header was normalized. Once in the database, we threaded messages by following the references and in-reply-to headers[3]. Unfortunately, the references and in-reply-to headers are not required in RFC standards, and many messages did not contain these headers. When these headers are missing, the email thread is broken, resulting in an artificially large number of small threads. For CTR and RTC, we reduced the number of broken threads from 18.3% and 15% to 7.7% and 1.8%, respectively[4].

4. ARCHIVAL DATA RESULTS

In this section, we present results related to our research questions. Since each question requires a different metric, we describe the metric and discuss any limitations of it in the section in which it is used. Although the metrics may be unfamiliar to readers, they are designed to produce similar measures to those used in traditional inspection experiments.

4.1 Frequency and Activity

Q1: Are developers able to maintain an adequate level of review during times of increased development activity?
We want to understand if developers can maintain a sufficient level of peer review during times of increased development. We measure the relationship between development activity and reviews. We examine the frequency as the number of reviews per month.

For **RTC**, review frequency is measured in two independent ways: by counting how many reviews received at least one reply and by counting how many commit logs contain the name of at least one reviewer in the "reviewer" field. For **CTR**, we count the number of commits that receive at least one reply.

Figure 1 provides a box and whiskers plot of each review type. The bottom and top of the box represent the first and third quartiles, respectively. Each whisker extends 1.5 times the interquartile range. The median is represented by the bold line inside the box. Our data are not normally distributed, so we report median values. The per month medians are 18, 16, and 19 for RTCA, RTC, and CTR respectively. We assume that commit activity is related to development activity. In order to determine the relationship between commit activity and the the review types, we conducted Spearman correlations – a non-parametric test. The only correlation above 0.50 is between the number of CTR contributions and the number of commits ($r = 0.69$)[5]. This correlation indicates that the number of CTRs changes proportionally to the number of commits. Therefore, when there is more code to be reviewed, there are more CTR reviews. This finding suggests that as the number of commits increases, CTR does not become, as one Apache developer feared, "commit-then-whatever". The number of RTCA contributions was weakly correlated with commits ($r = 0.14$), suggesting that the number of accepted contributions is only weakly related to commit activity. Surprisingly, the number of RTC contributions and the number of RTCA contributions was only moderately correlated ($r = 0.37$). These last two results suggest that the Apache group is very conservative in the contributions it accepts and is not influenced by high volumes of submissions. By reviewing a contribution around the time it is committed, Apache developers reduce the likelihood that a defect will become embedded in the software.

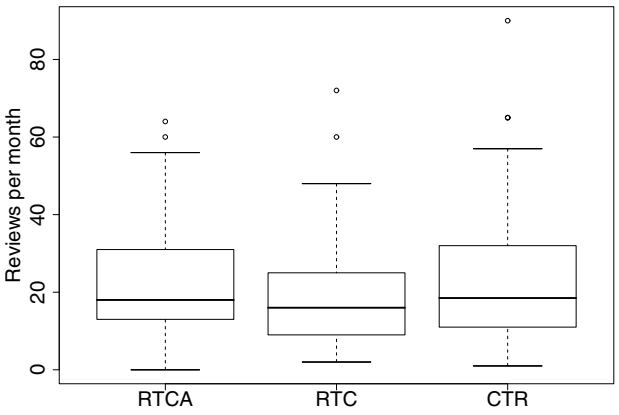

Figure 1: Number of reviews per month

4.2 Participation

Q2: How many reviewers respond to a review? How much discussion occurs during a review? What is the size of the review group?
It is simple to count the number of people that come to an inspection meeting. Ascertaining this metric from mailing-list-based reviews is significantly more difficult.

The first problem is that developers use multiple email addresses. These addresses must be resolved to a single individual. We use Bird et al.'s [2] tool to perform this resolution.

The remaining problems relate to the data available for each review type. RTCA provides a record of core-group members who performed a review. This measure does not

[3]For more information see RFC 2822 – Internet Message Format

[4]A detailed description of our technique is available at `helium.cs.uvic.ca/thread`

[5]All reported correlations are statistically significant at $p < 0.001$

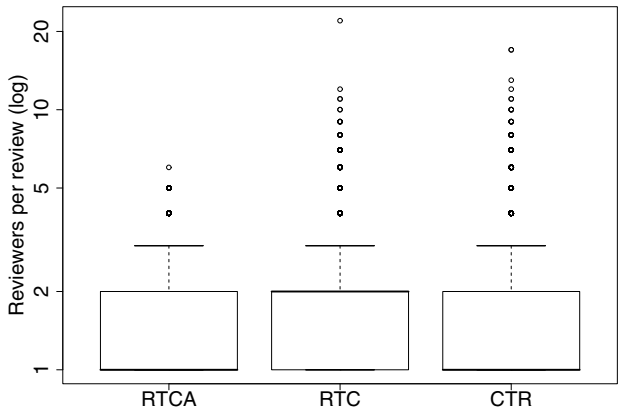

Figure 2: Number of reviewers per contribution

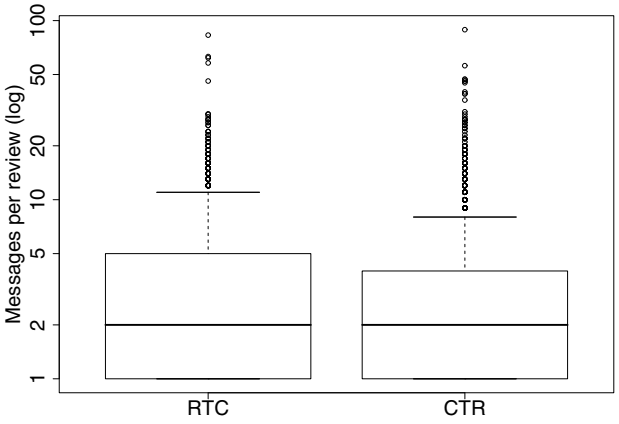

Figure 3: Number of messages per review

usually include developers who contributed reviews but are not part of the core-group. In contrast, RTC and CTR include all reviewers. However, it is only possible to count reviewers who respond to a contribution. So if an individual performed a review and did not find any issues or found the same issue as other reviewers, this individual would not be recorded as having performing a review. To overcome the latter limitation, we assume that if an individual is performing reviews over a long enough period of time, he or she will eventually be the first person to find an issue and will respond to a contribution (if a reviewer never responds, the reviewer is not helping the software team). We define the **review group** on a monthly basis as all individuals who responded to a review within a given month (i.e., number of reviewers per month). In summary, we have three metrics to gauge participation in reviews: the number of developers per review (roughly equivalent to the number of people who actively participate in an inspection – see Figure 2), the number of emails per review (the amount of discussion per review – see Figure 3), and the review group or the number of people who performed at least one review in a given month (roughly equivalent to the pool of reviewers who participate in inspections – see Figure 4).

Examining RTCA contributions, 50% of contributions are reviewed by only one individual and 80% are reviewed by two (see Figure 2). Apache policy states that three core-group members must review an accepted contribution. These results indicate that this policy is not always followed. Since only core-group members are recorded as reviewers in the commit log, RTCA has fewer recorded reviewers than RTC.

Figures 2 and 3 show that RTC has slightly more reviewers and messages in the discussion than CTR. For 80% of the contributions, RTC has three individuals and six messages, while CTR has two individuals and five messages. The median for RTC is two individuals and two messages, while the median for CTR is one individual and two messages. Interestingly, CTR has a larger number of responses from the author of the contribution than RTC. For CTR, the author and reviewer are often the only discussants (although others may be watching the discussion).

The review group (the potential number of reviewers, see Figure 4) is much larger than the number of reviewers per review – for RTC the median is 14, while for CTR the median is 15. The size of this review group is close to the number of core-group members found by Mockus et al. [14]. Developer expertise and specialization may account for the review group being much larger than the number of reviewers per contribution. In Fogel's [9] experience, developers often defer to an individual who has worked in a given area and proven his or her competence. In previous work, we have seen high levels of core-group (i.e., expert) participation in Apache reviews [23], as have Asundi and Jayat [1].

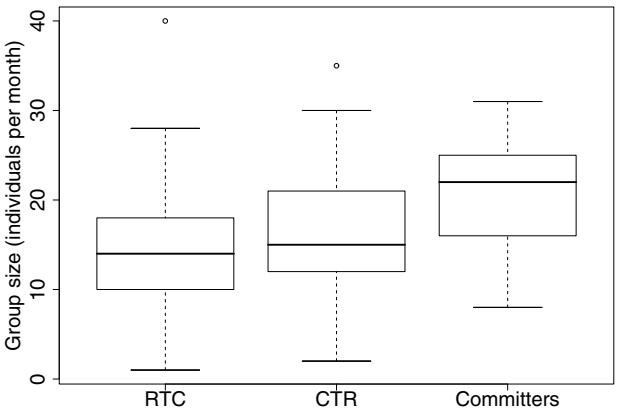

Figure 4: Review group

Although there are relatively few discussions with more than 15 messages and eight participants, there are rare threads with as many as 90 messages and 20 participants. Manual examination of larger threads revealed that some messages had nothing to do with the contribution. Instead, the contribution had started a debate on a larger issue. For example, one 89-message thread regarding a trivial contribution contained the entire debate on whether the CTR policy should be formally accepted.

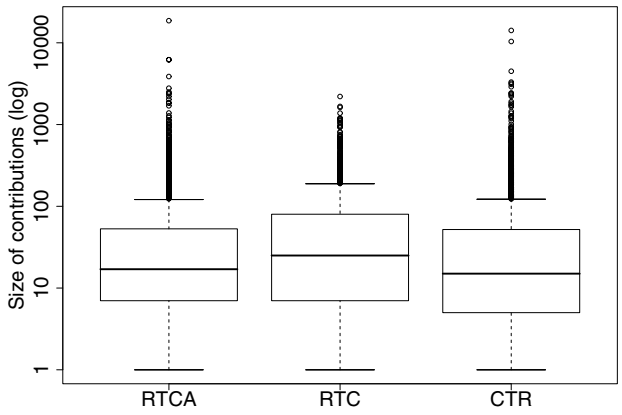

Figure 5: Size of contributions

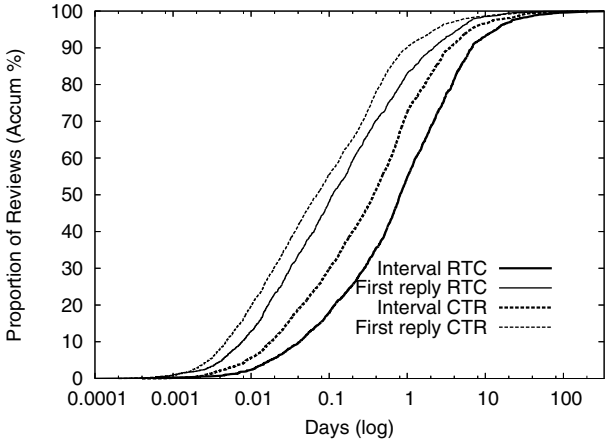

Figure 6: Review intervals. RTC takes longer than CTR

4.3 Size

Q3. Why is the size of the artifact under review so small? The size of the artifact under review is a common measure in the inspection literature [18]. Mockus et al. [14] found that changes to Apache were smaller than changes to the industrial projects they examined, but they could not explain why this was the case. We examine the size of changes that are reviewed. The change size is measured by summing the number of added and deleted lines.

Figure 5 shows the size of contributions for Apache reviews. **RTC** had the largest contribution size. The median number of changed lines was 25, and 80% of the contributions had less than 106 changed lines. RTC accepted, **RTCA**, contributions were smaller, with a median of 17, and 80% with less than 71 changed lines. The size difference between RTC and RTCA is consistent with our manual examination of contributions; often neophytes submitted contributions that contained solutions to multiple problems. These contributions were broken down into their components before being accepted. **CTR** had a slightly smaller contribution size with a median of 15 changed lines, and 80% of CTR contributions had less than 70 lines changed. These small contribution sizes are consistent with the Apache policy to review only small, complete, independent contributions.

4.4 Review Interval

Q4. What is the calendar time to perform a review? Porter et al. [20] define review interval as the calendar time to perform a review. The full interval begins when the author prepares an artifact for review and ends when all defects associated with that artifact are repaired. The pre-meeting interval, or the time to prepare for the review (i.e., reviewers learning the material under review), is also often measured. Figure 7 provides a pictoral representation of review interval.

RTC. For every contribution submitted to the mailing list, we determine the difference in time from the first message (the "[PATCH]") to the final response (the last review or comment). Figure 6 shows the cumulative logarithmic distribution of the review intervals. The bottom line in the figure represents the full review interval for RTC. The figure shows that 80% of the contributions are reviewed in less than 3.8 days and that 50% are reviewed in less than 19 hours. We also determine the time to the first response. This time is roughly equivalent to what is traditionally called the pre-meeting time. The preparation time for the contributor is negligible, since it is a single command to create a diff. This first-response time is slightly less than one day for 80% of the contributions and has a median of three hours. Only 9% of the RTC reviews last for more than one week, indicating that initially ignored contributions are rarely reviewed.

CTR. Since the contribution is committed before it is reviewed, we need to know not only the time between the commit and the last response (the full CTR review interval, see Figure 6), but also the time between the commit and the first response. This first response indicates when the issue was discovered; the ensuing discussion may occur after the problematic contribution is removed. Ideally, the amount of time defective, unreviewed code is in the system should be short. The first review happens very soon after a commit: 50% of the time it occurs within 1.7 hours and 80% of the time it happens within 11 hours. The discussion or full interval of the review lasts longer, with a median value of 8.9 hours. In 80% of cases, the discussion lasts less than 1.7 days. Only 5% of issues are found after one week has passed. Like RTC, this final result indicates that if a change is not reviewed immediately, it will likely not be reviewed.

The data indicates that reviews, discussion, and feedback happen very quickly. The following quotation from a mailing list discussion in January 1998 supports these findings.

> `I think the people doing the bulk of the committing appear very aware of what the others are committing. I've seen enough cases of hard to spot typos being pointed out within hours of a commit.`

4.5 Defects

Q5. How many reviews find defects? We know from previous work that Apache likely has defect rates that are comparable to those in industry [14, 22]. We measure how many reviews find defects.

Ideally, we would be able to use the traditional measure of the total number of defects found during review. Since this information is based on discussion, it is subjective and must

be recorded manually. Unfortunately, Apache developers do not record the number of defects found in a review. We use random sampling to manually determine how many reviews contain at least one defect. The language used in the reviews makes it apparent whether at least one defect was found; however, without being actively involved in the review, it is difficult to determine how many defects in total were found per review (the traditional measure of defects).

We categorized reviews as containing a defect or not containing a defect. We further categorized defects as being at the source code level or being abstract. Abstract defects included architectural and design issues. A single review could contain both a source and an abstract defect.

RTC. From a random sample of 100 reviews, we found that 51% contained at least one defect. Of the reviews that did contain defects, 84% contained a source code defect, 63% of reviews contained an abstract defect, and 47% contained both types of defects.

CTR. In a random sample of 100 reviews, 66% contained at least one defect. Of the reviews that did contain defects, 64% contained a source code defect, 62% contained an abstract defect, and 30% contained both types of defects.

A common question is, "do these defect results imply that half of the data on which these previous figures are based really has nothing to do with defects, but is some other kind of discussion?" Not all reviews are expected to find defects, especially when the review is of a very small artifact, such as in the case of Apache (see Section 4.3). Of the 200 threads we examined, it was rare that a thread did not qualify as a review. An example of a review that did not qualify as a "true" review, is the policy discussion referenced in Section 4.2.

Since the contribution sizes are very small (see Section 4.3), one would expect that the discussion would remain very localized. However, our findings indicate that a large proportion of the reviews that found defects discussed the abstract or global implications of the contribution. One explanation for this finding is the lack of "code ownership" exhibited by Apache developers (i.e., Apache developers have a holistic view of the system) [14].

Interestingly, while manually examining the reviews, it became apparent that the fix for a defect was discussed immediately and a new solution was usually proposed and implemented. The new implementation sometimes came from one of the reviewers.

4.6 Comparison of RTC to CTR

The goal of a software process is to produce high quality software in a timely manner. Review techniques can be compared based on their effectiveness (e.g., how many defects they find) and efficiency (e.g., how long it takes to find and remove those defects). We compare RTC to CTR based on these two metrics.

The main reason for adopting a CTR policy is that RTC slows down development by increasing the review interval. The frustration with RTC is apparent in the policy discussion regarding the acceptance of CTR in January of 1998 (see Section 2).Our first hypothesis relates to the efficiency of the review techniques. We hypothesize that *CTR has a shorter review interval than RTC.*

With CTR, the contribution is committed before it is reviewed, so if no issues are found in review, the interval is zero (i.e., the author does not have to wait for the review before committing the contribution). In this case, it is obvious that CTR has a shorter interval than RTC – using the median value, RTC is 19 hours slower than CTR (see Section 4.4). However, if an issue is found, a discussion must occur and the contribution must be fixed or removed. In this case, the median interval for CTR is 8.9 hours. To determine if there is a statistically significant difference between the two review types, we run a Kolmogorov-Smirnov test with the null hypothesis that RTC and CTR have the same interval (i.e., the two distributions are not statistically significantly different). Since $p < 0.001$, we reject the null hypothesis and conclude that CTR has a shorter interval than RTC – using the median value, CTR is 2.2 times faster than RTC. In both cases, CTR has a shorter interval than RTC.

We have determined that CTR is more efficient than RTC. However, if CTR finds fewer defects than RTC, it is a less effective review technique. We hypothesize that *CTR finds fewer defects than RTC.*

Ideally, we would compare the number of reviews that do not find defects to the number of reviews that do find defects. However, this comparison would produce a result biased in favor of CTR because Apache policy requires RTC reviewers to respond with positive reviews, while only negative responses are required for CTR (see Section 2). We create an unbiased measure by calculating the number of reviews that have a question from the reviewer, but do not contain a defect. Since asking a question always requires the reviewer to send a message, there should be no bias towards either review technique. From our manual examination of reviews (see Section 4.5), we know that for RTC there are 51 reviews that find defects to 7 reviews with questions and no defects found, while for CTR the ratio is 66 to 9.

We perform a Chi-squared test on the defect counts, with the null hypothesis being that RTC finds the same number of defects as CTR. Since $p = 0.80$, we conclude that there is no statistically significant difference between the number of defects found for the two review techniques. Although we cannot reject the null hypothesis and we do not have an exact measure of the number of reviews that do not find defects, the evidence that we have supports the conclusion that RTC and CTR find a similar number of defects.

In summary, we have found that while CTR has a shorter review interval than RTC, the two techniques likely find the same number of defects. Although these results indicate that, in the Apache project, CTR is a superior review technique to RTC, we must remember the different contexts in which they are used by the project (see Section 2). CTR is only used when a trusted, core-group member feels confident in what they are committing, while RTC is used when the core-group member is less certain or by developers without commit privileges. In an industrial environment, RTC could be applied to new hires, incurring a longer review interval, while CTR could apply to developers who have shown that they are competent. An experiment controlling for other variables (e.g., developer expertise, complexity of artifacts under review) may lead to CTR being used in a wider context.

5. COMPARISON OF APACHE PEER REVIEW TO INSPECTION AT LUCENT

Siy has provided us with the data collected from an inspection experiment conducted at Lucent Technologies. These

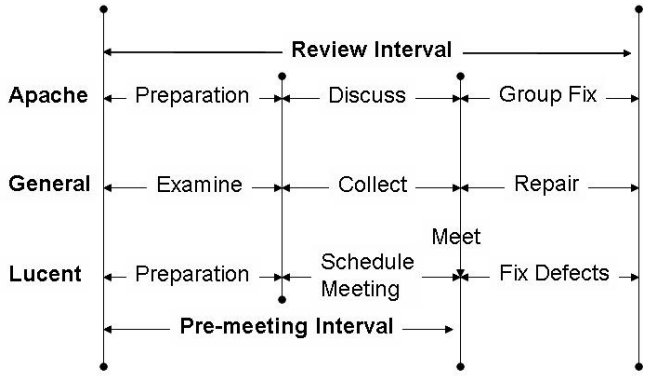

Figure 7: The typical stages involved in a peer review. One Lucent inspection is the equivalent of 5.5 Apache reviews. The median pre-meeting interval is 10 days for Lucent and 19 hours for Apache RTC. Asynchronous reviews reduce the amount of time lost to scheduling and allow for more detailed discussions.

data have been used in a number of studies [20, 19, 18]. The software is a compiler that consists of 55 kLOCS of C++; the most recent version of Apache is written mostly in C and is 110 kLOCS[6]. Although the projects are not identical, we feel that a preliminary comparison is appropriate.

The number of defects does not provide a useful comparison between projects. Weller [25] reports a 7 to 1 difference in the number of discovered defects in two separate projects using the same inspection technique – different projects using distinct techniques are likely to produce even more dissimilar results. For example, a more complex project will likely have higher levels of defects regardless of the review technique used. As a result, we cannot comment on the effectiveness of the Apache review techniques compared to those used at Lucent.

Interval is likely less affected by project differences (e.g., code complexity and developer skill level), since the time a developer spends performing the review is always much smaller than the actual review interval regardless of the technique (e.g., no developer [except perhaps the author] spends days or weeks solely working on an inspection).

We must first normalize the size of the artifact as the difference is substantial. Lucent inspections have a median artifact size of 265 Non-Comment Source Lines (NCSL). In Section 5, we measure size as the number of lines changed in a contribution. To provide a fair comparison, we must only look at source code and count the total number of non-comment lines involved in the review. We also include diff context, as the reviewer will likely examine this as well. For RTC contributions, we found a median size of 48 NCSLs. Normalizing for the size of the artifact, we find that one Lucent contribution is 5.52 times the size of an RTC contribution. We cannot be certain that Apache reviews include the time to fix the defect, so we must compare the full Apache review interval as calculated in Section 4.4, 19 hours or .8 days, to pre-meeting at Lucent, 10 days (instead of the 15 days it takes to perform the review and fix all the defects at Lucent). Combining the two results we have the following: for an RTC to cover the same amount of code as a Lucent inspection, we must perform 5.5 RTCs, which takes 4.4 days. Therefore, RTC is 56% faster than Lucent inspection.

Given the many differences between these projects and the difficulties in comparing effectiveness, we do not want to make strong claims about the efficiency of the Apache review techniques. However, we suggest that the differences in review interval are not unreasonable. First, while keeping the number of defects constant, Perry et al. [17] were able to reduce the inspection interval by 25% simply by conducting asynchronous inspections (i.e., by eliminating the time wasted in scheduling meetings, the middle section in Figure 7). Apache reviews are also conducted asynchronously. Second, Apache reviews likely reduce the time required for the preparation stage (the first section in Figure 7). As noted by Mockus et al. [14], Apache developers do not have "code ownership" (i.e., they work in many different sections of the software). This lack of "code ownership" increases the likelihood that reviewers will already be familiar with the section of the artifact related to the contribution under review. Furthermore, frequent, small contributions require developers to constantly review changes and thus keep up-to-date with the activities of their peers. This familiarity with the software should reduce the amount of extra preparation necessary to conduct a review. Fourth, through manual examination, we found that the discussion usually switches from defect finding to defect fixing (the last section in Figure 7). Occasionally, one of the reviewers would provide a fix to a contribution under review. This group problem solving could lead to a decrease in the time to fix defects. The new solutions created within the review are immediately re-reviewed before being incorporated into the software product.

6. TOWARDS A THEORY OF OSS PEER REVIEW

This case study allowed us to investigate our research questions and report results for each. The research questions addressed

- the review process, policies, and structure,
- the frequency of reviews and relationship between the frequency of reviews and development activity,
- the level of participation in reviews and the size of the review group,
- the size of the artifact under review,
- the review interval (i.e., the calendar time to perform a review), and
- the number of reviews that find defects.

Case studies can provide an excellent background from which to develop new theories. One cannot generalize from a single case, but the findings of a single case study can inform the selection of future case studies as well as the design of controlled experiments. In this section, we restate some of our case study findings as components of a theory, and suggest why each component may lead to a successful peer review technique. The following statement encapsulates our understanding of how Apache code review functions.

[6]See `http://www.koders.com` Accessed September 2007

(1) Early, frequent reviews (2) of small, independent, complete contributions (3) conducted asynchronously by a potentially large, but actually small, group of self-selected experts (4) leads to an efficient and effective peer review technique.

We dissect this statement below, showing the evidence that we have gathered, how this evidence is related to existing literature, and the evidence that needs be obtained through future work.

1. Early, frequent reviews
The longer a defect remains in an artifact, the more embedded it will become and the more it will cost to fix. This rationale is at the core of the 30-year-old Fagan inspection technique [7]. We have seen comparatively high review frequencies for Apache in Section 4.1. Indeed, the frequencies are so high that we consider Apache review as a form of "continuous asynchronous review". We have also seen a short interval that indicates quick feedback (see Section 4.4).

2. of small, independent, complete contributions
Breaking larger problems into smaller, independent chunks that can be verified is the essence of divide-and-conquer. Morera and Budescu [15] provide a summary of the findings from the psychological literature on the divide-and-conquer principle. Section 4.3 illustrates the small contribution size in Apache, as does the work of Mockus et al. [14], and Section 5 provides a comparison of contribution sizes with an industrial project. The policy discussion in Section 2 provides support for the idea that Apache developers will review only independent, complete solutions.

Although there are many advantages to small contributions, one potential disadvantage is fragmentation. Industrial development may require assurances that all aspects of a product have been reviewed before a release. A system for tracking these small contributions would make this accountability possible and eliminate concerns about unreviewed code: "commit-then-whatever".

3. conducted asynchronously by a potentially large, but actually small, group of self-selected experts
The mailing list broadcasts the contribution to a potentially large group of individuals. A smaller group of reviewers, approximately 15, performs reviews periodically. An even smaller group of reviewers, between one and two, actually participates in any given review (see Section 4.2). Small contribution sizes coupled with self-selection should allow the number of individuals per review to be minimized. The larger a contribution, the more likely it is to require a broad range of expertise. It is possible to increase the level of expertise in a group by adding more individuals to the review [24]. Optimizing the number of reviewers based on the complexity and size of a contribution could lead to overall cost savings.

Two papers [1, 23] indicate that a large percentage of review responses are from the core group (i.e., experts). Fielding [8], Raymond [21], and Fogel [9] discuss how OSS developers self-select the tasks they work on. One risk with self-selection is that it is possible that no one will review the contribution. If self-selection of reviews were used in industry, a review that was not naturally selected could be assigned to a developer.

In a large or poorly modularized project a broadcast mechanism would likely become too busy (i.e., developers would be interested in a small proportion of the information). One possible solution is to limit the broadcast to developers working on related modules. However, the advantage of a broadcast is that developers outside of the core-group can contribute when the core-group is lacking the required expertise for a particular problem.

4. leads to an efficient and effective peer review technique
We have shown that Apache review is efficient in Section 5. Although we were unable to compare Apache review to inspection in terms of effectiveness, we found, through random sampling in Section 4.5, that CTR and RTC both find defects. Given that previous research has shown that Apache has a comparable defect density to industrial software [22, 14], it is reasonable to assume that peer reviews play a part in this success.

The number of defects found is a limited measure of effectiveness [10]. Apache reviews provide opportunities for the other benefits of peer review, such as education of new developers, group discussion of fixes to a defect, and abstract or wandering discussions. With traditional inspections, the discussion centers around defects. A good mediator does not allow reviewers to start discussing anything but defects [26]. In contrast, asynchronous reviews ease the time constraints placed on the review meeting, allowing for a more open discussion.

7. REPLICATIONS

As discussed in the introduction, we chose to examine a single project in order to provide sufficient depth and detail. As a result, the most pressing area of our future work is replication of this case study. Without replication, the case study and any theory derived from it may be only applicable to the original case (i.e., to Apache). Yin [27] identifies two types of case study replications: literal replications and contrasting replications. The purpose of a literal replication is to ensure that similar projects produce similar results. For example, Subversion has adopted an Apache style of review, and it would serve as an excellent literal replication [9]. Contrasting replications should produce contrasting results, but for reasons *predicted* by the theory. For example, a recent study at Cisco used an RTC technique, but assigned reviewers to particular contributions [3]. This Cisco case would be an ideal contrasting replication for testing the efficacy of allowing reviewers to self-select the artifacts they feel competent to review. By performing both types of replications, the theory is tested and strengthened.

8. CONCLUSION

We have described two lightweight review techniques: CTR and RTC. We are unaware of any paper that has empirically examined a CTR style of peer review. With the exception of policy and review participation, we are unaware of any paper that has empirically examined an OSS RTC style of peer review.

This study advances our understanding of the Apache development process. We provide a theory and study methodology that have the potential to advance our understanding of OSS review in particular and peer review in general. It is possible that aspects of this theory could be incorporated into industry. For example, dividing reviews into smaller, independent, complete pieces may reduce the burden placed on any individual reviewer and divide the often unpleas-

ant review task into more manageable chunks that can be conducted periodically throughout the day. These changes might result in industrial developers taking a more positive view of peer review. Lussier provides an example of an OSS-style RTC technique being successfully modified for use in an industrial setting [13].

It is unlikely that any peer review technique will be clearly superior in all environments. As an example, further experimentation may find that although RTC and CTR have significantly shorter intervals when compared to inspection, formal inspection finds more defects than either RTC or CTR. An optimal process may be created by performing faster reviews early in the development process and by formally inspecting critical sections of code infrequently before major releases. Combinations that incorporate the most successful aspects of various software processes, regardless of their origin, are more likely to produce high quality software in a timely manner.

9. ACKNOWLEDGMENTS

The authors would like to thank Harvey Siy for providing inspection data from the Lucent inspection experiments and for his feedback on an earlier draft of the paper, Laura Cowen for checking our statistical analyses, Chris Bird for allowing us to use his email name aliasing tool, and Laura Young for providing editorial support. Rigby would also like to acknowledge support from an NSERC CGSD.

10. 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, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proceedings of the 2006 International Workshop on Mining software repositories*, pages 137–143, 2006.

[3] J. Cohen. *Best Kept Secrets of Peer Code Review*. Smart Bear Inc., Austin, TX, USA, 2006.

[4] T. Dinh-Trong and J. Bieman. The FreeBSD Project: A Replication Case Study of Open Source Development. *IEEE Transactions on Software Engineering*, 31(6):481–494, 2005.

[5] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proc. of the 18th ACM symposium on Operating Systems Principles*, pages 57–72, 2001.

[6] M. Fagan. *A history of software inspections*. Springer-Verlag, Inc., New York, NY, USA, 2002.

[7] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):182–211, 1976.

[8] R. T. Fielding and G. Kaiser. The Apache HTTP Server Project. *IEEE Internet Computing*, 1(4):88–90, 1997.

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

[10] P. M. Johnson. Reengineering inspection. *Commun. ACM*, 41(2):49–52, 1998.

[11] J. Lawrence G. Votta. Does every inspection need a meeting? *SIGSOFT Softw. Eng. Notes*, 18(5):107–114, 1993.

[12] G. Lee and R. Cole. From a Firm-Based to a Community-Based Model of Knowledge Creation: The Case of the Linux Kernel Development. *Organization Science*, 14(6):633–649, 2003.

[13] S. Lussier. New tricks: how open source changed the way my team works. *Software, IEEE*, 21(1):68–72, 2004.

[14] A. Mockus, R. Fielding, and J. Herbsleb. A Case Study of Open Source Software Development: The Apache Server. *Proceedings of the 22nd International Conference on Software Engineering*, pages 262–273, 2000.

[15] O. Morera and D. Budescu. A Psychometric Analysis of the Divide-and-Conquer Principle in Multicriteria Decision Making. *Organizational Behavior and Human Decision Processes*, 75(3):187–206, 1998.

[16] Netcraft. Web server survey. `http://news.netcraft.com/archives/2006/04/06/april_2006_web_server_survey.html`, Accessed September 2006.

[17] D. Perry, A. Porter, M. Wade, L. Votta, and J. Perpich. Reducing inspection interval in large-scale software development. *Software Engineering, IEEE Transactions on*, 28(7):695–705, 2002.

[18] A. Porter, H. Siy, A. Mockus, and L. Votta. Understanding the sources of variation in software inspections. *ACM Trans. Softw. Eng. Methodol.*, 7(1):41–79, 1998.

[19] A. Porter, H. Siy, C. Toman, and L. Votta. An experiment to assess the cost-benefits of code inspections inlarge scale software development. *Software Engineering, IEEE Transactions on*, 23(6):329–346, 1997.

[20] A. A. Porter, H. P. Siy, and J. Lawrence G. Votta. Understanding the effects of developer activities on inspection interval. In *Proc. 19th International Conference on Software Engineering*, pages 128–138, 1997.

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

[22] Reasoning, Inc. Apache defect and metric reports. `http://www.reasoning.com/downloads.html`, 2003.

[23] P. C. Rigby and D. M. German. A preliminary examination of code review processes in open source projects. Technical Report DCS-305-IR, University of Victoria, January 2006.

[24] 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.

[25] E. F. Weller. Using metrics to manage software projects. *Computer*, 27(9):27–33, 1994.

[26] K. E. Wiegers. *Peer Reviews in Software: A Practical Guide*. Addison-Wesley Information Technology Series. Addison-Wesley, 2001.

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