Noname manuscript No.
(will be inserted by the editor)

An Analysis of Merge Conflicts and Resolutions in Git-based Open Source Projects

Hoai Le Nguyen · Claudia-Lavinia Ignat

Received: date / Accepted: date

Abstract Version control systems such as Git support parallel collaborative work and became very widespread in the open-source community. While Git offers some very interesting features, resolving conflicts that arise during synchronization of parallel changes is a time-consuming task. In this paper we present an analysis of concurrency and conflicts in official Git repository of four projects: Rails, IkiWiki, Samba and Linux Kernel. We analyse the collaboration process of these projects at specific periods revealing how change integration and conflict rates vary during project development life-cycle. We also analyse how often users decide to rollback to previous document version when the integration process generates conflicts. Finally, we discuss the mechanism adopted by Git to consider changes made on two continuous lines as conflicting.

Keywords Version control systems · Parallel work · Conflicts

1 Introduction

Computer supported collaboration is an increasingly common occurrence that becomes mandatory in academia and industry where the members of a team are distributed across organizations and work at different moments of time. Version control systems are popular tools that support parallel work over shared projects and offer support for synchronization of parallel changes on those projects. Version control systems can be classified into centralised version control systems (CVCSs) and decentralised version control systems (DVCSs).

Hoai Le Nguyen
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
E-mail: hoai-le.nguyen@inria.fr

Claudia-Lavinia Ignat
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
E-mail: claudia.ignat@inria.fr

CVCSs such as CVS (Berliner, 1990) and Subversion (Collins-Sussman et al., 2004) rely on a client-server architecture. The server keeps a complete history of versions while clients keep only a local copy of the shared documents. Users can modify in parallel their local copy of the shared documents and synchronize with the central server in order to publish their contributions and make them visible to the other collaborators.

DVCSs such as Git (2005), Mercurial (2005) and Darcs (2003) became popular around 2005. These systems rely on a peer-to-peer architecture where each client keeps the history of versions plus a local copy of the shared documents. Users can work in isolation on their local repositories. They can also synchronize their local repository with the ones of other collaborators.

Even if CVCSs are largely used in academia and industry they feature some limitations due to their centralised architecture: they offer a limited scalability and limited fault tolerance, administration costs are not shared and data centralisation in the hands of a single vendor is an inherent threat to privacy. On the other side, DVCSs overcome these limitations of centralised architectures. Their peer-to-peer architecture allows them supporting a large number of users and tolerating faults. Each user keeps a copy of the history and decides with whom to share it without storing it on a central server. These features made DVCSs widely used in the domain of open software development where projects have often a large number of contributors.

Studies showed that in large projects the partition of software modules among developers is limited and developers can contribute to any part of the code (Gutwin et al., 2004). In Git, users can synchronize their changes with other users working in parallel with them. In this process, a merge is performed between local changes and remote changes. Similar to other commercially available merge tools, Git uses a textual merging technique (Mens, 2002) where software artefacts, i.e. files, are viewed as flat text files. Merging approaches are line-based, where lines of text are considered as indivisible units. If concurrent changes refer to the same file, we say that these changes are conflicting. Conflicts on a file can be automatically resolved or they need user intervention for their resolution. We call the former category automatically resolved conflicts and the latter category unresolved conflicts. If conflicting changes refer to different non-adjacent lines of the file, the conflict is automatically resolved by the system. If, on the contrary, conflicting changes refer to the same or adjacent lines of the file, the conflict cannot be automatically resolved and the user has to manually resolve it. Unresolved conflicts also occur if a file is renamed and modified/deleted concurrently, if it is modified and deleted concurrently, if it is renamed concurrently by two users, if a user renames a file with the same name as another user gives to a concurrently created file and if two users concurrently add two files with the same name. Note that by the name of a file we understand the whole path identifying that file.

Conflicts are costly as they delay the development process (de Souza et al., 2003). In the period of time between conflicts occur and they are discovered and understood, they might grow and become difficult to resolve. Developers

may postpone integrating parallel work as they fear that conflicts may be hard to resolve. The potential conflicts concern makes parallel work to diverge more and conflicts to more likely to happen and grow.

Understanding how often and when conflicts are more likely to happen during the development process and how users resolve them can help proposing awareness mechanisms that can prevent conflicts from happening. This study could help proposing better merging approaches that minimize conflicts that users have to manually resolve. In this paper we study traces of projects developed with Git in order to quantitatively analyse the different types of textual conflicts at the level of files that arise at the different development cycle phases. One particular type of unresolved conflict that we study is that referring to adjacent lines. If concurrent changes occur on two adjacent lines Git signals an unresolved conflict, but not in the case of two lines separated by two or more lines. As there is no reason why these cases are treated differently, we aim studying whether developers resolve them differently. We also aim quantitatively measuring merge user satisfaction after a conflict resolution in terms of how often users roll back to a previous version. Even if several existing studies focused on parallel changes and conflicts on Git-based projects (Brun et al., 2013; Kasi and Sarma, 2013), they did not analyse fine-grained conflicts at file level and their resolution mechanisms.

Several tools rather than using textual merging, use syntactic or semantic merging (Mens, 2002). Syntactic merging takes the syntax of software artefacts into account, while semantic merging considers semantic information. Studies such as Brun et al. (2013) and Kasi and Sarma (2013) considered both textual conflicts as result of textual merging and higher order conflicts that are conflicts at semantic level that cause compilation errors or test failures. Other studies such as de Souza et al. (2004) studied indirect conflicts when changes to one software artefact affect concurrent changes to another artefact. In de Souza et al. (2004) authors proposed the social call graph that describes dependencies between software developers for a piece of code. The social call graph combines the call graph data structure that contains all the dependency relationships of a software application with authorship information. Our study does not investigate indirect and higher level conflicts and focuses uniquely on conflicts related to the same file with a particular attention for textual conflicts as used by main DVCSs such as Git.

The rest of the paper is organized as follows. In Section 2 we explain briefly related works. Section 3 presents our conflicts measurement during the merge process. Section 4 discusses implications for design for our study and some limitations of analysing Git repositories. Section 5 gives some concluding remarks.

2 Related work

In this section we present some studies on parallelism of changes performed in version control systems.

The user study presented in Reiher et al. (1994) reports on conflict resolution experiences with the optimistic file system Ficus. Conflicts were classified into *update/update*, *remove/update* and *naming* conflicts. *Update/update* conflicts appear when two concurrent updates are performed on the same file. *Remove/update* conflicts appear when an update of a file and the removing of that file were performed concurrently. A *naming* conflict occurs when two files are independently created with the same name. The study found out that only about 0.0035% of all updates made to non-directory files resulted in conflicts and among them less than one third could not be resolved automatically. Authors mentioned that conflicts that cannot be resolved automatically are any *update/update* concurrent changes on source code or text files as they have arbitrary semantics and therefore require user intervention. Note that in contrast to the definition of conflicts used in Reiher et al. (1994), in the terminology of version control systems two updates done on the same file (source code or textual) lead to non-automatically resolved conflicts only if the updates refer to the same or adjacent lines in the file. All *update/remove* conflicts required human intervention and about 0.018% of all *naming* conflicts led to name conflicts which have to be resolved by humans.

In Perry et al. (2001), the authors presented a study about parallel changes in the context of a large software development organization and project. The study analyses the complete change and quality history of a subsystem of the Lucent Technologies' 5ESS over a period of 12 years. Each set of change requests representing all or part of a solution to a problem was recorded by the system. When a change from this set was made on a file, the system kept track of the lines added, edited or deleted. This set of changes composes a *delta*. It was found that 12.5% of all *deltas* were made to the same file by different developers within a day. 3% of all these *deltas* made within a day by different developers physically overlap. However, interference of these *deltas* is analysed over a quite large period of time (1 day) and not all these *deltas* are performed concurrently.

In Zimmermann (2007) authors investigated four large open-source projects (GCC, JBoss, JEdit and Python) and found that in CVS the *integration rate* that measures the percentage of concurrently modified files over all modified files is very low (between 0.26% and 0.54%). The study found that the *conflict rate*, i.e. the proportion of files with unresolved conflicts over concurrently modified files, varies between 23% and 47%. Low integration rates indicate that the parallel changes within single files are rare and have small impact to the development process. High conflict rates indicate that parallel changes affect the same location within a file or can not be integrated automatically by CVS.

Only few studies analysed parallel changes and conflicts for projects developed using DVCSs. In Brun et al. (2013) and Kasi and Sarma (2013), authors studied the *merge conflict rate* of merges for several open-source projects using Git repositories. They found that the average merge conflict rate was 16%. However, these studies did not analyse the types of conflicts during merge and the frequency of conflicts at file level as measured in Zimmermann (2007).

Also these studies did not analyse quantitatively what are the resolution mechanisms adopted by users.

In Brindescu et al. (2014) analysed how centralised and distributed version control systems influence the practice of splitting, grouping and committing changes. However, this study did not analyse merge commits that represent decisions on conflict resolution. Our study is mainly focused on studying conflicts and therefore we analysed developers merging behaviour through merge commits.

In McKee et al. (2017) authors propose a qualitative study on the factors that impact how practitioners approach merge conflicts and the difficulties they face when resolving conflicts. The study was conducted based on semi-structured interviews on 10 software practitioners across 7 organizations. The study found that the factors that mostly describe merge conflict difficulty are complexity of conflicting lines of code, the knowledge/expertise in area of conflicting code, the complexity of the files with conflicts and the number of conflicting lines of code. Our study is complementary to McKee et al. (2017) and studies quantitatively textual conflicts inside a file and investigates different aspects such as their frequency, their length and their localisation and the ways developers resolve those conflicts such as by maintaining concurrent changes or rollbacking to previous code versions. It is important to understand what types of conflicts arise at the different moments during the lifetime of a project. This can help tool designers and developers to choose the different tools and techniques that can be applied during the lifetime of a project.

In conclusion, no study quantitatively analysed fine-grained conflicts inside files in DVCSs and how people resolve these conflicts.

3 Measurements

In order to measure the level of parallelism and the proportion of conflicting modifications in DVCSs, we adopted an experimental methodology where we analysed the corpus of four large open-source projects developed using Git:

- **Ruby on Rails** (Ruby, 2015) is a web framework, with integrated support for unit, functional, and integration testing. We analysed version 5.0.0.alpha of the Rails.
- **IkiWiki** (IkiWiki, 2015) is a wiki software system that compiles wiki pages into HTML pages for publication. We analysed IkiWiki version 3.0.
- **Samba** (Samba, 2015) is an implementation of networking protocols to share files and printers between Unix computers and Windows computers. We analysed Samba 3.0.x.
- **Linux Kernel** (LinuxKernel, 2015) is an implementation of a Unix-like computer operating system kernel. We analysed version 4.x of the Linux kernel.

Beside the large size and the popularity of these projects, they are representative for the different software development pull-based (Gousios et al., 2014)

models that they adopt. In practice, the core-development-team will organize at least one repository as the primary repository where the latest approved changes can be found. Contributors can clone from this official repository. However, only the core-development-team has the write-access to commit directly. Other contributors need to use the pull-based development model in which a contributor creates a pull-request for his changes. A core-team's member then inspects the changes and decides to pull and merge contributor's changes to the main repository or not. And in some cases, contributors are requested to update or add more changes before their pull-request is accepted. Nowadays, the pull-based model is naturally supported by web-based hosting services such as GitHub (2008) and Bitbucket (2008).

Rails project uses pull-request model which is naturally supported by GitHub (2008). Contributors can fork (clone) from the official GitHub repository and contribute via GitHub's pull-requests. In a pull-request, reviewers and its contributor communicate directly using pull-request's comments. These comments are available to other users and they can participate into this conversation. Afterwards a pull-request can be merged to the main line or declined.

IkiWiki looks like a *private repository* where contributors send their *patches* to Josey Hess, the main developer of the project. Samba uses a *shared repository* among registered contributors. It uses an auto-build system for code-review process. Contributors need to join a technical mailing-list before contributing. Linux Kernel uses a pull-based model via mailing-list. Contributors need to send their *patches* to the appropriate subsystem maintainer's mailing-list in charge of the different parts of the project.

Table 3 presents some details about these projects: the period of their development (until 05-October-2015), the number of commits, the number of contributors (authors), the number of created files during the lifetime of the project and the number of existing files on 05-October-2015. Note that if a file is moved during the lifetime of the project from a place to another, we counted it as a new created file.

Project name	*Period (days)*	*No. of commits*	*No. of authors*	*No. of created files*	*No. of existing files*
Rails	3,967	19,375	3,422	10,272	2,984
IkiWiki	3,496	53,625	982	4,610	3,362
Samba	7,094	100,301	386	33,626	7,582
Kernel	5,132	547,515	14,395	90,173	51,567

Table 1 Open source projects developed using Git

In contrast to CVCSs, Git does not support the centralized logging feature of all user activities. The best overview of user activities is provided by the commit history (including merges) from the primary repository. To identify concurrences and conflicts in each project, we created a *shadow* repository

and recursively re-integrated developer's changes into this repository. In other words, by means of Python scripts (GitChangesAnalyzer, 2017) we re-played all merges that were performed during the development period of each project.

3.1 Integrations and conflicts on files

We first determined the number of concurrent updates to a same file and then the number of concurrent updates to a same file that resulted in unresolved conflicts. Similar to Zimmermann (2007) we computed the *integration rate* and *conflict rate* as provided in Table 2. *File updates* represents the total number of updates to files. A file can be updated several times throughout the development cycle. *Integration rate* represents the proportion of concurrent updates to a same file over all updates to files. *Conflict rate* is calculated by the proportion of updates to a same file that resulted in unresolved conflicts over concurrent updates to files. The file updates were collected from all commits of the project. And by re-integrating all developer's changes, we computed the concurrent updates to a same file and the concurrent updates to a same file that resulted in unresolved conflicts.

Project name	*File updates*	*Integration rate*	*Unresolved conflict rate*
Rails	117,960	4.04%	16.26%
IkiWiki	37,327	1.08%	50.50%
Samba	306,182	0.68%	87.84%
Kernel	1,278,247	10.99%	4.86%

Table 2 Concurrency and conflicts on files

We can notice that Kernel and Rails projects have larger integration rate than IkiWiki and Samba. For instance, integration rate in Kernel project is 10 times larger than IkiWiki and 16 times larger than Samba. This can be explained by the large size of Kernel project in terms of the number of files. In contrast with the *integration rate*, Rails and Kernel have smaller *conflict rates* than IkiWiki (50.50%) and Samba (87.84%). We do know that Rails is a large project using advantages of GitHub, which supports pull-based model naturally. GitHub interface allows not only the author of a pull-request and the reviewer but also other contributors and core-team members to discuss about that pull-request and its issues. It brings a big advantage of sharing collaborators knowledge to solve problems during integration. In case of Linux Kernel, it uses pull-based model via mailing list with a list of subsystem maintainers. It also has a list of delegated servers, such as *linux-next*, where commits are tested before they are pushed to primary repository (German et al., 2015). On the other side, Samba uses shared repositories among contributors and

IkiWiki is maintained as a private repository by Josey Hess. Nowadays, all of them provide a list of Todo tasks and a list of Bugs where contributors can focus their work to avoid conflicting integration.

The lack of a central server that holds a reference copy of the project introduces more parallelism between user versions allowing them to diverge more in DVCSs than in CVCSs. For instance, Kernel, Rails, IkiWiki and Samba projects developed in Git have significantly (99% confidence level) higher *integration rate* (22, 8, 2 and 1.5 times respectively) than projects in CVS analysed in Zimmermann (2007). However, the higher *integration rate* does not result into higher *conflict rate*. For instance, Kernel and Rails have 5 and 1.5 times respectively lower *conflict rate* than projects in CVS whereas Samba and IkiWiki have almost 2 times higher *conflict rate*. The *conflict rate* in Git's projects depends on collaboration process management.

Project name	*Content* conflict	*Remove/Update conflict*	*Naming conflict*
Rails	89.68%	2.97%	7.35%
IkiWiki	46.31%	1.48%	52.22%
Samba	64.47%	34.44%	1.09%
Kernel	90.96%	8.63%	0.41%

Table 3 Proportion of conflict types

We also measured the proportion of the different conflicts types: content conflicts referring to conflicts inside a file, remove/update conflicts referring to concurrent removal and update of a file and naming conflicts referring to concurrent renaming of the same file or of two files with the same name. Table 3 presents the proportion of conflict types of four projects. We found that content conflict is far the most popular type of conflict with a proportion of 46% - 90% from all conflict types.

3.2 Integrations and conflicts based on release dates

In the previous section we presented the *integration rate* and *conflict rate* of four projects over their whole development period. However during their development life-cycle, activities are not equally distributed. Our hypothesis is that collaborative activities achieve some peaks around project release dates, such as periods of one or two weeks before a release date. To gain a better understanding about collaboration during those active periods, we conducted an analysis about integrations and conflicts based on project release dates.

Figure 1 illustrates four active periods of one week length before and after respectively the release date (RD). We denote these periods as follows: B2W (between two weeks before RD and one week before RD), B1W (between one

week before RD to RD), A1W (between RD to one week after RD) and A2W (one week after RD to two weeks after RD). We also analysed B4W, B3W, A3W and A4W.

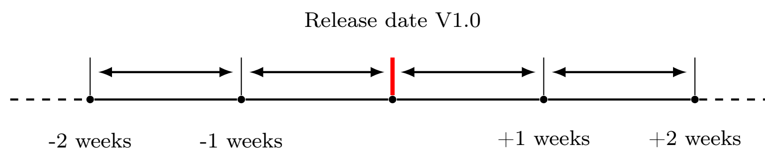

Fig. 1 Analysis based on release date

3.2.1 IkiWiki

There are three official versions of IkiWiki, however the first two versions did not introduce any concurrency nor conflict. In fact, at that time, IkiWiki was developed by Josey Hess only. We analysed the latest version V3.0 with a release date on 31-12-2008. The result is presented in Figure 2.

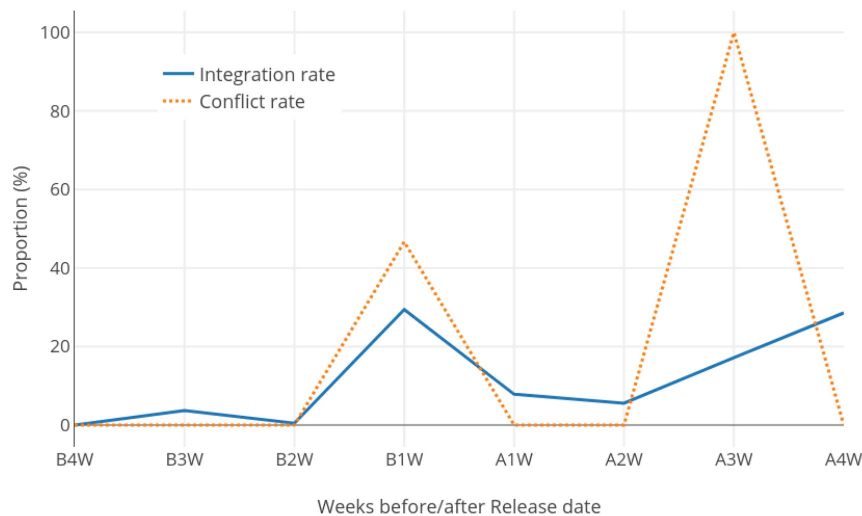

Fig. 2 IkiWiki-V3.0, integration and conflict rate on files

The result shows that one week before RD, the *integration rate* is very high (29.41%) with a conflict rate of 46.67%. Also, the *integration rate* decreases in the next two weeks after RD(A1W, A2W) and increases in the third and the fourth weeks after RD (A3W, A4W). All integrations in A1W, A2W and A4W are successful and all integrations in A3W generated conflicts.

This behaviour can be explained by the policy adopted by IkiWiki where contributions are merged by Josey Hess just one week before RD. Figure 3, extracted from IkiWiki official site (IkiWiki, 2015), is an illustration for this explanation. Many merges and commits are submitted close to RD. There are still some minor integrations such as bug-fixing after RD. We also analysed the following four weeks after A4W (A5W to A8W) and did not find any integration nor conflict. So we can say that for IkiWiki, the integration of version V3.0 begins one week before the official release date and lasts four weeks after that release date.

2008-12-31	Joey Hess	formatting
2008-12-31	Joey Hess	fix link
2008-12-31	Joey Hess	Merge branch 'master' of ssh://git.ikiwiki.info/srv...
2008-12-31	Joey Hess	Merge branch 'next'
2008-12-30	intrigeri	Merge commit 'upstream/master' into prv/po
2008-12-29	Joey Hess	update
2008-12-29	Joey Hess	Merge branch 'master' into next
2008-12-28	Joey Hess	Merge branch 'master' into next
2008-12-26	Joey Hess	Merge branch 'master' into next
2008-12-26	Joey Hess	Merge branch 'master' into next
2008-12-25	Joey Hess	Merge branch 'master' into next
2008-12-25	Joey Hess	more 3.0 docs, changelog
2008-12-24	Joey Hess	typo

Fig. 3 IkiWiki-V3.0, commits in the week before RD

3.2.2 Samba

Samba community started using Git as its main repository from version 3.2. We chose to analyse three Samba versions: 3.2, 3.3 and 3.6 based on the number of merges between B4W and A4W. In fact we could not find any merges in the period (B4W-A4W) in other versions (4.0, 4.1, 4.2, 4.3). The results are presented in Figure 4 and Figure 5.

Figure 4 shows that the *integration rates* of version 3.2 and 3.3 are almost similar. They have high *integration rates* in the week before RD, in the second week after RD and in the fourth week after RD. These *integration rates* are 2 times (for V3.2, B1W), 6 times (for V3.2, A2W), 3 times (for V3.3, B1W) and 4 times (for V3.3, A2W) respectively higher than the overall *integration rate* (0.68%, Table 2).

The integration process in version 3.6 changed compared to versions 3.2 and 3.3. Version 3.6 had integrations only in the two weeks before RD and the first week after RD (B2W and A1W). Furthermore, its *integration rate* is three to six times lower than in versions 3.2 and 3.3.

In Figure 5, we can see that some integrations in version 3.2 and 3.3 in A2W period resulted in conflicts and all other integrations in B4W-A4W period are successful. Specially version 3.6 does not introduce a single conflict during B4W-A4W. In fact, starting from version 3.6, each release version officially

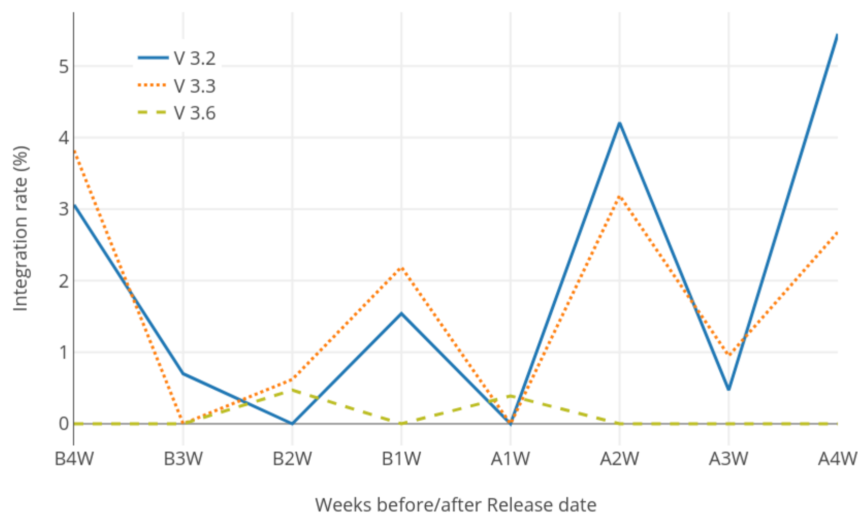

Fig. 4 Samba, integration rate on files

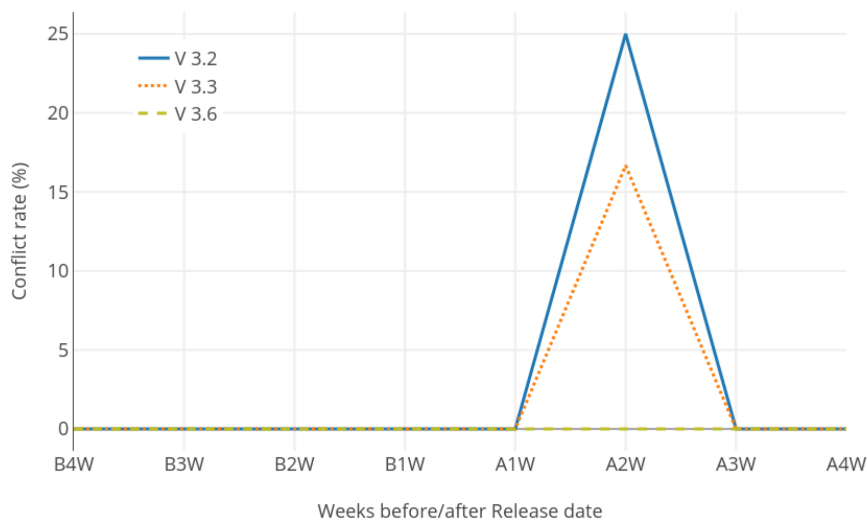

Fig. 5 Samba, conflict rate on files

has a series of pre-release (pre) versions and release-candidate (rc) versions. For version 3.6, it has three pre versions (3.6pre1, 3.6pre2, 3.6pre3) and three rc versions (3.6rc1, 3.6rc2, 3.6rc3). It means that most collaboration works are done in pre and rc versions, not in official release version 3.6. It also explains why we could not find any merges in B4W-A4W of other versions after 3.6.

3.2.3 Rails

The first official version (1.0) of Rails was released in 13-December-2005. However, the most active periods started from version 3.1. We chose to analyse the following Rails versions: 3.1, 3.2, 4.0, 4.1 and 4.2. The results are presented in Figure 6 and Figure 7.

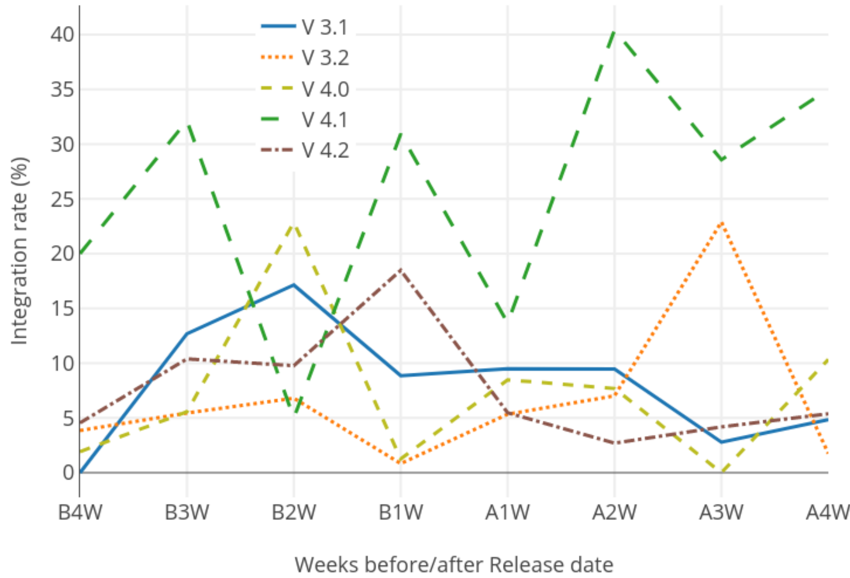

Fig. 6 Rails, integration rate on files

Versions 3.1 and 4.2 have high *integration rates* in B3W-A3W periods. Version 3.2 and 4.0 have two distinct active periods of integration: B3W-B2W and A1W-A3W. Version 4.1 has three distinct active periods of integration: B4W-B3W, B1W-A2W and A4W. Generally, for versions 3.2 and 4.0 the integration was done in the two weeks before the release date (B3W-B2W). Their *integration rates* in B1W are very low: 0.83% (V3.2) and 1.26%(V4.). However, only version 3.2 is conflict-free in B1W while version 4.0 has a high *conflict rate* (30%). In contrast, versions 4.1 and 4.2 have high *integration rates* in B1W: 30.93% and 18.48% respectively for V4.1 and V4.2. They also have high *conflict rates* in B1W: 30.93% (V4.1), 23.08%(V4.2). We can see that the integration process in Rails had changed from version 3.1 with a high *integration rate* in B1W to versions 3.2 and 4.0 with a very low *integration rate* in B1W. Then the *integration rate* became very high in versions 4.1 and 4.2.

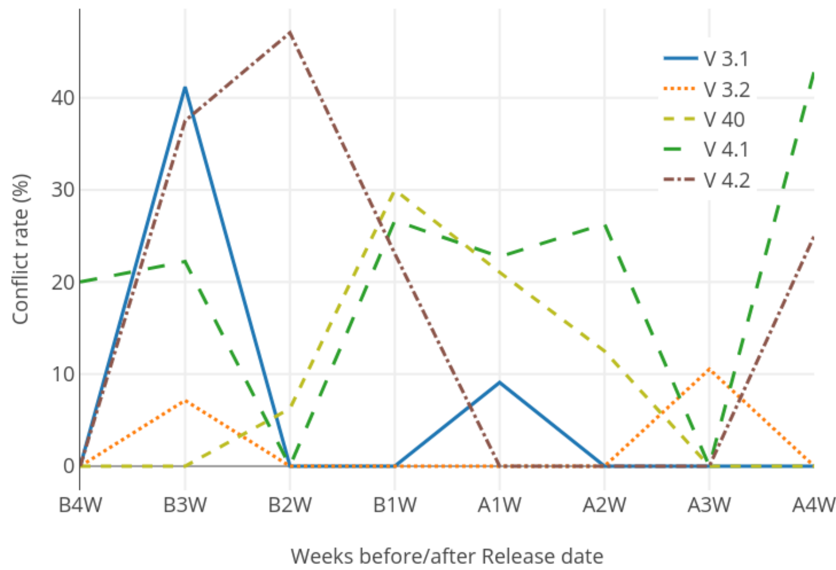

Fig. 7 Rails, conflict rate on files

3.2.4 Linux Kernel

Linux Kernel community started using Git as the main repository in version 2.6.x. We chose to analyse the versions 2.6.39, 3.0, 3.1, 3.19, 4.0 and 4.1. Each version was developed in 2-3 months. The versions 2.6.39, 3.0, 3.1 and 3.19 were released in 2011 while 4.0 and 4.1 were released in 2015. We analysed how the collaboration process in Linux Kernel was changed over years.

Figure 8 shows that with the exception of V3.19, the five other versions have almost similar integration trends with an active period of integration of three weeks after RD(A1W-A3W). It has 4-8 times higher *integration rate* than other periods. For version 3.19, the active period of integration is two weeks before RD(B2W-B1W) with a 7 times higher *integration rate*.

In fact, Linux Kernel community uses a development process called 'merge windows'. At the beginning of a new version development cycle - right after the official release date of previous version - the 'merge window' is opened. The bulk of changes is merged during this time. The 'merge window' lasts for approximately two weeks. After this period a series of release-candidates (rc) are proposed over the next six to ten weeks. On this time, only patches which fix problems are allowed to be submitted to the mainline. This explains why in Linux Kernel, the most active period of integration is A1W-A3W. Note that all the changes integrated during the merge window have been collected, tested and staged ahead of time. It keeps the *conflict rate* of Linux Kernel in this period quite low (3.6%-8.39%) in comparison to active periods of other projects such as Rails (40%, A2W), Samba (25%, A2W), IkiWiki (46%, B1W).

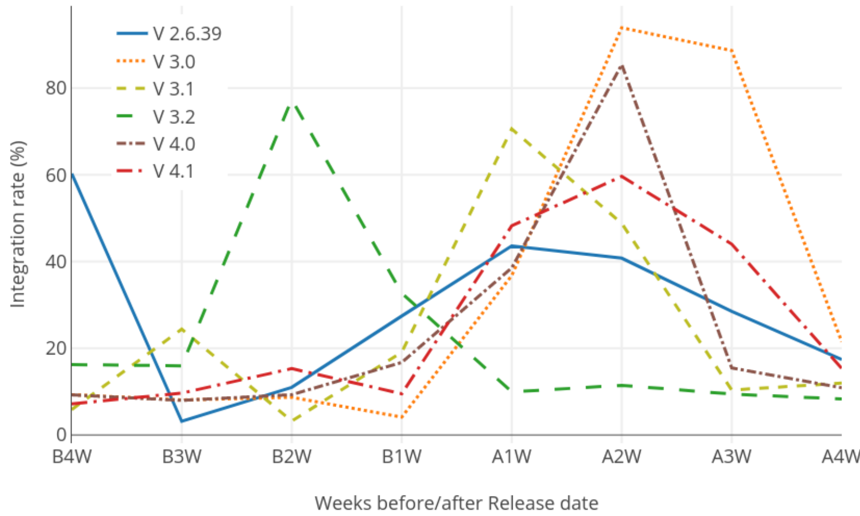

Fig. 8 Linux Kernel, integration rate on files

Additionally, Figure 9 shows that *conflict rate* in B2W is sightly higher than in A2W although A2W is the most active period of integration. We can explain that integration before the RD has higher chance to result into conflicts.

3.2.5 Overall

By analysing the collaboration process of these projects at specific time periods which are close to release dates, we can reveal how change integration evolves over time. In IkiWiki (3.0), Samba (3.2, 3.3) and Rails (3.1), an 'old integration style' was found in which most integration works were submitted just one week before release date (B1W). This style was changed in the next versions of Samba (3.6) and Rails (3.2, 4.0) that is integration works were submitted two weeks before release date. Moreover, the integration process of Rails was changed also in its next versions (4.1, 4.2). This time, it became worst with very high *integration rate* in B1W (2.5-3.5 time higher than in version 3.1). Linux Kernel, a special project, has a more stable integration process which is called 'merge window'. This integration process was changed sightly over its versions. With this analysis, we can know when Samba community started using pre-release and release-candidate versions and how the 'merge window' is used effectively in Linux Kernel. However, not all the stories are revealed such as why version 3.19 has a different active period (B2W) in comparison with other Linux Kernel versions.

Table 4 presents the proportion of conflict types based on Release date. Our analysis revealed that content conflict *'CONFLICT (content)'* is the most

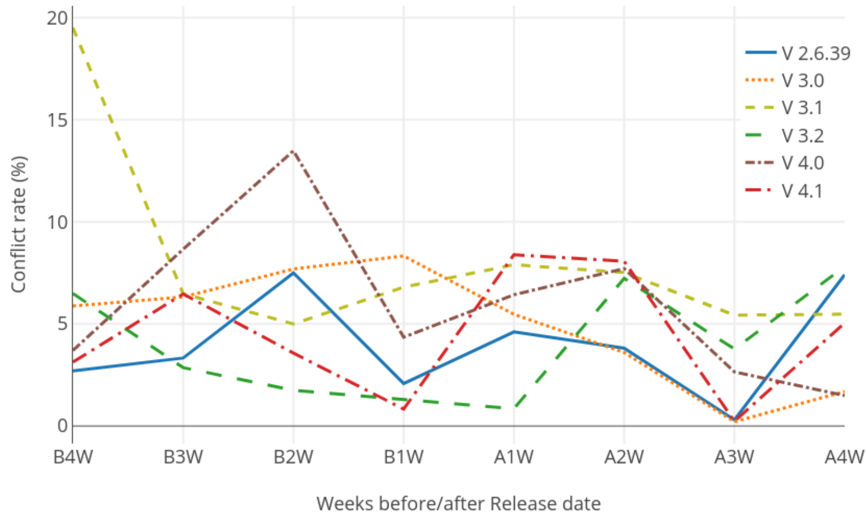

Fig. 9 Linux Kernel, conflict rate on files

popular conflict type with 93.3%, 100%, 76%-100%, 81.85%-97.41% over all conflicts respectively in IkiWiki, Samba, Rails and Linux Kernel.

In addition, we measured the Spearman's rank correlation coefficient (Rho) (SpearmanRankCorrelation, 2018) between integration rate and conflict rate. We collected data points based on the release date. The result which is presented in Table 5 shows that the correlation between them is none or very weak in both Linux (48 data points, negative monotonic) and Rails (18 data points, positive monotonic). We do not have enough valid data points for IkiWiki and Samba.

3.3 Conflict resolution

Several files can be in conflict during a merge. We counted the total number of merges performed during the lifetime of the project and the number of merges that leaded to unresolved conflicts. Table 6 gathers these results of all projects analysed. When a merge is not resolved automatically by Git, users need to resolve it manually. A user can decide to rollback to a previous version. We provide also the rollback rate, i.e the number of times user uses rollback action over all the merges that contain unresolved conflicts. The rollback rate is lower in Kernel and Rails compared to IkiWiki and Samba.

In practice, after a successful merge, users build and test the merging results. A successful merge can result in build-failed or test-failed. In order to be considered as a *successful integration*, a successful textual merge needs to be built and tested successfully also. Otherwise, it is considered as *higher-order*

Project name	*Content* conflict	*Remove/Update conflict*	*Naming conflict*
Rails	89.68%	2.97%	7.35%
IkiWiki	46.31%	1.48%	52.22%
Samba	64.47%	34.44%	1.09%
Kernel	90.96%	8.63%	0.41%
IkiWiki 3.0	93.33%	6.67%	0.005%
Samba 3.2	100.00%	0.00%	0.005%
Samba 3.3	100.00%	0.00%	0.005%
Samba 3.6	0.00%	0.00%	0.005%
Rails 3.1	62.50%	0.00%	37.505%
Rails 3.2	100.00%	0.00%	0.005%
Rails 4.0	100.00%	0.00%	0.005%
Rails 4.1	100.00%	0.00%	0.005%
Rails 4.2	76.00%	4.00%	20.005%
Linux 2.6.39	96.12%	3.88%	0.005%
Linux 3.0	97.41%	2.59%	0.005%
Linux 3.1	81.85%	17.74%	0.005%
Linux 3.19	97.26%	2.74%	0.005%
Linux 4.0	95.16%	4.84%	0.005%
Linux 4.1	94.49%	4.72%	0.005%

Table 4 Conflict types based on release date

Project name	*No. of* data points	*Rho*	*P-value*
Rails	18	0.102	0.343
IkiWiki	2	NA	NA
Samba	2	NA	NA
Kernel	48	-0.188	0.1

Table 5 Spearman's rank correlation coefficient of integration rate and conflict rate

conflict (Brun et al., 2011). To detect these conflicts, we have to build using provided project build-scripts and test using provided test-sets every successful merge. We did not measure these types of conflicts as not all projects provided test-sets.

We provided in Table 7 a more detailed analysis of the rollback action in which we find rollback actions in all merges, not only the ones that contain textual-conflicts. *Level* represents the number of following commits after a merge when the rollback actions happen. Figure 10 illustrates *levels* of rollback in which branch *B2* is merged to branch *B1*. The merge result is *M0* and *C1, C2, C3* are the following commits after the merge.

Normally, if a user decides to rollback, he can use the *'git revert'* command to rollback the merging. The *'git revert SHA-1-B1'* usually creates a new commit after the merging commit (i.e commit *C1* has the same *SHA-1 hash* with *B1*). We assigned *level=1* to this case. However, in the most of the cases,

Project name	*No. of merges*	*Unresolved conflict merge rate*	*Rollback rate*
Rails	9,728	4.34%	1.66%
IkiWiki	1,037	7.52%	5.13%
Samba	1,281	10.07%	6.98%
Kernel	38,961	9.11%	0.70%

Table 6 Frequencies of conflicting merges

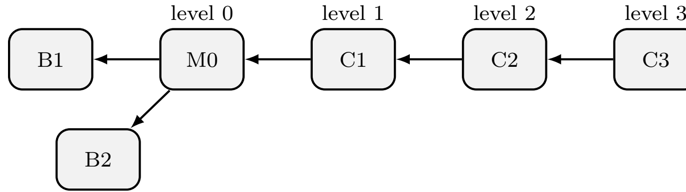

Fig. 10 Levels of rollback action

Project name	*Level 0*	*Level 1*	*Level 2*	*Level 3*	*Level 4*
Rails	3,217	36	2	0	0
IkiWiki	39	13	1	1	0
Samba	260	2	0	0	0
Kernel	2,016	1	0	0	0

Table 7 Rollback action after merging

users don't want to create new commits. Users can use a set of commands to rollback without creating a new commit such as *'git checkout SHA-1-B1; git reset –soft HEAD; git commit -amend'* that will rollback *M0* to *B1* version (i.e commit *M0* has the same *SHA-1 hash* with *B1*). We assigned *level=0* to this case. Note that, in practice, when merging a contributor's work to the main repository, the core-team members can use *'git merge -s ours'* to chose the main-line version as the default result when conflicts happen or use *'git merge –no-commit'* to test the merging results and manually fix the conflicts before committing them. Furthermore, the rollback actions can happen in the next one, two or three commits. We assigned *level=1, level=2, level=3* to these cases. In our measurements, we limit *level* to four. In fact, we could not find any rollback actions after three commits from the merging result.

Table 7 shows that Rails (33.46%) and Samba (20.45%) have many more rollback actions than IkiWiki(5.21%) and Kernel(5.18%).

3.4 Adjacent-line conflicts

In this section we analyse Git mechanism of considering concurrent modifications of two continuous lines as being in conflict (called adjacent-line conflicts). Figure 11 illustrates an example of adjacent line conflict with both expected and real merge result. User at site-1 makes changes on line 2 and user at site-2

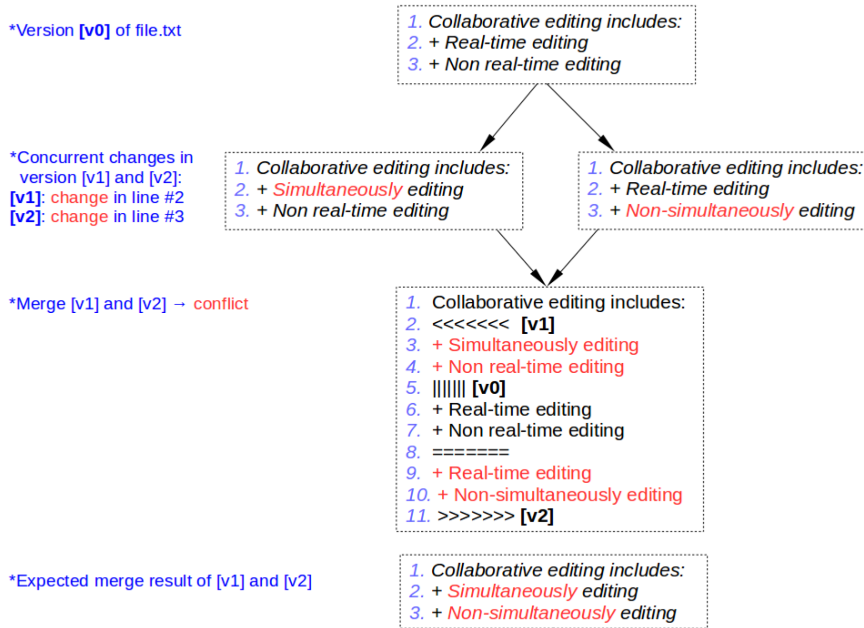

Fig. 11 Adjacent-line conflict: expected and real merge results

makes changes on line 3. They then merge their changes and Git generates a *'CONFLICT (content)'*. Our hypothesis is that this is not a content conflict because these changes are made on two different lines. Git should merge them successfully by applying changes from both sites. To test our hypothesis, we analysed all content conflicts in the four projects to detect all adjacent-line conflicts. We then analysed the adjacent-line conflict resolutions that were manually fixed by the authors to check if both changes done on the adjacent lines were applied. If in most cases both changes were applied we can conclude that adjacent-line conflicts should not be considered conflicts.

We used *'git difftool'* to detect the changed lines of two sites in comparing to the original document. Figure 12 illustrates how the changed-lines is detected by denoting a changed line as 'X' and an unchanged line as 'V'. After updating the changed-lines for each *content conflict*, we used adjacent-line patterns for each two continuous lines. Figure 13 (Group 1) illustrates

adjacent-line conflicts where the first line is changed by only one site and the second line by only the other site.

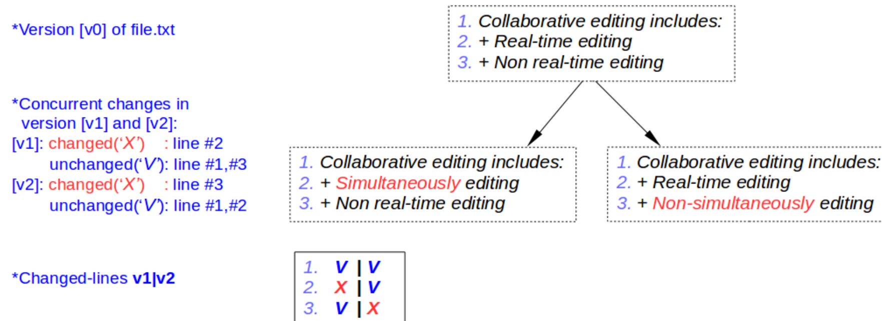

Fig. 12 Changed-lines

Figure 13 (Group 2) presents four patterns including both an adjacent-line conflict and a normal content conflict. In our analysis of adjacent-line conflicts, we excluded these patterns by considering that they can be resolved as normal content conflicts. In our analysis we considered only adjacent-line conflicts which do not include normal content conflicts.

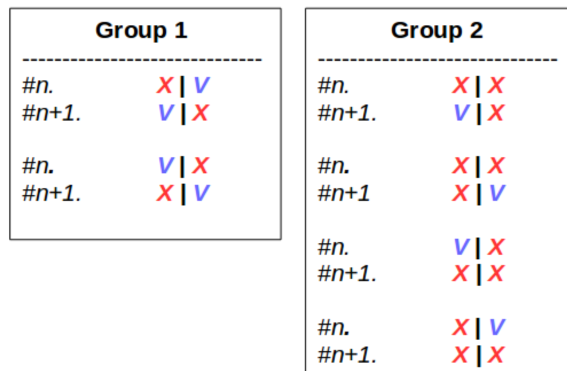

Fig. 13 Adjacent-line patterns

The results are presented in Table 8 illustrating the number of adjacent-line conflicts and their resolutions. Three types of resolutions exist: applying both changes, applying a change from one site only (either from site-1 or site-2) and applying no changes.

The proportions of applying both site changes are 24.39% in Samba, 57.5% in Rails, 75% in IkiWiki and respectively 85.01% in Linux Kernel. We did

Project name	*Adjacent -line conflicts*	*Applying both changes*	*Applying one change*	*Other cases*
Rails	80	46	28	6
IkiWiki	4	3	1	0
Samba	41	10	23	8
Kernel	367	312	51	4

Table 8 Adjacent line conflicts and resolutions

	Adjacent-line conflicts		**Normal content conflicts**	
Project name	*No. of conflicts*	*Applying both changes*	*No. of conflicts*	*Applying both changes*
Rails	80	57.50%	317	5.67%
IkiWiki	4	75.00%	22	9.09%
Samba	41	24.39%	1149	14.19%
Kernel	367	85.01%	1326	13.38%

Table 9 Adjacent-line and normal content conflicts

the same analysis on how users resolve normal content conflicts and Table 9 presents a comparison of the results obtained for adjacent-line conflicts and normal-content conflicts.

We compared the frequency of applying both changes for adjacent-line conflicts to that of applying both changes for normal content conflicts. Table 10 presents for each project the standardized mean-difference effect size (SdMD) between proportions of applying both changes for adjacent-line and normal content conflicts (EffectSizeCalculator, 2017). Only Samba has a low SdMD and its lower bound of confidence interval is less than zero. Excepting Samba, we obtained significant confidence at level of 95% that ‘adjacent-line conflicts’ are resolved more often by applying both changes than ‘normal content conflict’. Applying both changes in the case of a conflict means that the concurrent changes were not in conflict, so the conflict was not necessary to be detected.

We had conducted an empirical test on adjacent-lines changes for Darcs and SVN (GitChangesAnalyzer, 2017). The results show that SVN (CVCS) and Darcs (DVCS) merge changes in adjacent lines successfully.

4 Discussion

We found that content conflict is far the most popular type of conflict with a proportion of 46% - 90% from all conflict types. Compared to centralised version control systems such as CVS, in Git we have a significantly higher

Project name	*SdMD*	*Lower 95% confidence interval*	*Upper 95% confidence interval*
Rails	1.8872	0.4319	3.6909
IkiWiki	2.0614	1.4932	2.2812
Samba	0.405	-0.0384	0.8483
Kernel	2.1837	1.9854	2.382

Table 10 Standardized mean difference(SdMD) effect size between adjacent-line conflicts and normal content conflicts

integration rate, i.e. concurrent changes that refer to the content of the same file, varying from 1.5 to 22 times more than in CVS reported analysis. These integration rates are not equally distributed over the life time of the project but are higher close to release dates. In order to prevent conflicts, close to release dates developers should use awareness mechanisms about the location of their changes. Awareness, defined by Dourish and Bellotti (1992) as an "understanding of the activities of others which provides a context for your own activity", has been identified by the CSCW community as one of the most important issues in support of remote collaboration. For instance, the awareness approach proposed in Dewan and Hegde (2007) could be adopted to provide developers with warning messages concerning concurrent activity and the possibility to consult a list of conflicts. Based on a selected conflict, a user can set watches for concurrently edited elements. For instance, they can be notified when a collaborator has finished editing the element. Another solution could be annotation of concurrent changes (Ignat and Oster, 2008). However, this solution is suitable only for centralised version control systems that rely on a server and consists of computing divergence between the project local version of a user and the current state of the project that is saved on the server. Computing divergence in a distributed version control system where there is no central server remains a challenge. Solutions relying on Conflict-free Replicated Data Types such as André et al. (2013); Yu et al. (2015) that have been adopted by Atom (AtomTeletype, 2017) can be used. Moreover, the fact that the change in integration process can be identified based on the integration rate can be of use to other researchers trying to extract process related information from open-source projects.

We found that across the studied four repositories around 75% of the adjacent-line conflicts were false positives. Adjacent-line conflicts detection was designed to warn the developers that there are two changes on adjacent lines that might be related and that developers should check whether the changes are conflicting. For around 25% of cases adjacent-line conflict warnings indeed helped developers to discover the conflicts reducing the costs in later development phase. However, for the other 75% developers did some unneeded extra work for solving the conflicts. Moreover, if concurrent changes occur on two adjacent lines Git signals a conflict, but not in the case of two

lines separated by two or more lines. Our results suggest that Git should reconsider signalling conflicts on adjacent lines inside the source code file which requires developers to do in most of the cases some extra work for removing the conflict. A solution would be that Git sends a warning message to the users in the case of concurrent changes on adjacent lines, but does not represent this conflict inside the source code file. Note that in Subversion and Darcs version control systems, concurrent modifications on adjacent lines are not considered as conflict. Our suggestion would be useful for tool builders to help developers in avoiding wasting time on trivial merge conflicts.

Our study features some pitfalls of mining DVCS repositories as they lack a centralized logging server. The history may be rewritten by the repository owner (Bird et al., 2009): the order of commits can be altered, commits can be removed, a sequence of commits can be flattened from multi-branches into a single branch, edits can be squashed in multiple commits via *rebase*. In our analysis we ignored merges flattened due to rebasing. In German et al. (2015) the authors presented a new method called *continuous mining*. Instead of mining only the primary repository (called *blessed*), this method continuously observes all known repositories of a software project to cover the complete history of the project development. Their empirical study focuses on Linux Kernel (LinuxKernel, 2015) which has 479 repositories (from 2012). Among these repositories, 22% did not contribute a single commit to the *blessed*. Nevertheless, *continuous mining* is still a re-active logging mechanism and it is not considered as a permanent solution to the need of centralized logging features of Git.

By analysing only the history from official servers of four projects, we also missed information regarding user communication. The short description of commits does not include this data. Email threads in project's mailing-list such as Linux Kernel Mailing List (LinuxKernelMailingList, 2017) and GitHub conversations (comments of a pull-request or an issues) are promising for extracting this data.

5 Conclusion

The goal of this work was to analyse concurrencies and conflicts in Git, a decentralized version control system. We analysed the corpus of four large projects in Git and ours findings are as follows.

Generally, a higher *integration rate* of a project does not generate a higher *unresolved conflict rate*. It depends on how the integration process is managed. Linux Kernel is a large-scale project with a list of subsystem maintainers which keep the lowest *conflict rate* for this project although its *integration rate* is much more higher than the other projects. Excepting Linux Kernel, Rails which was developed using GitHub pull-based model has 5 times lower *conflict rate* than Samba(shared repository) and 3 times lower than IkiWiki(private repository). A more detailed analysis based on release dates shows that each

project has its own integration process with dynamic integration rates that changed during project's development cycle.

The rollback action is used more often in case of higher-order conflicts than in case of textual-conflicts. Most of rollback actions do not create new commits (*level=0*) meaning that users generally try to test the merge result before deciding to rollback or not.

In contrast to Darcs and SVN, Git considers changes made on two adjacent lines as content conflict. Based on our analysis, users mostly apply all changes when resolving the adjacent lines conflicts. We proposed that Git should merge concurrent changes on two adjacent lines and throw a warning message instead of considering them as conflicting.

References

André, Luc; Stéphane Martin; Gérald Oster; and Claudia-Lavinia Ignat (2013). Supporting Adaptable Granularity of Changes for Massive-scale Collaborative Editing. In E. Bertino, D. Georgakopoulos, M. Srivatsa, S. Nepal and A. Vinciarelli (eds): *CollaborateCom'13. Proceedings of the 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, Austin, Texas, USA, 20 October 2013 – 23 October 2013.* Washington: IEEE Computer Society, pp. 50–59.

AtomTeletype (2017) Code together in real time with Teletype for Atom. https://blog.atom.io/2017/11/15/code-together-in-real-time-with-teletype-for-atom.html. Accessed 19 March 2018.

Berliner, Brian (1990). CVS II: Parallelizing Software Development. In: *USENIX'90. Proceedings of the Winter 1990 USENIX Technical Conference, Washington, D.C, USA, 22 January 1990 – 26 January 1990.* Berkeley: USENIX Association, pp. 341–352.

Bird, Christian; Peter C. Rigby; Earl T. Barr; David J. Hamilton; Daniel M. German; and Prem Devanbu (2009). The promises and perils of mining git. In: *MSR'09. Proceedings of the 6th International Working Conference on Mining Software Repositories, Vancouver, BC, Canada, 16 May 2009 – 17 May 2009.* Washington: IEEE Computer Society, pp. 1–10.

Bitbucket (2008) Atlassian Bitbucket. *Code, Manage, Collaborate.* https://bitbucket.org/product. Accessed 19 March 2018.

Brindescu, Caius; Mihai Codoban; Sergii Shmarkatiuk; and Danny Dig (2014). How do centralized and distributed version control systems impact software changes? In P. Jalote, L. Briand, and A. van der Hoek (eds): *ICSE'14. Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May 2014 – 7 June 2014.* New York: ACM, pp. 322–333.

Brun, Yuriy; Reid Holmes; Michael D. Ernst; and David Notkin (2011). Proactive detection of collaboration conflicts. In: *ESEC/FSE'11. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on*

Foundations of Software Engineering, Szeged, Hungary, 5 September 2011 – 9 September 2011. New York: ACM, pp. 168–178.

Brun, Yuriy; Reid Holmes; Michael D. Ernst; and David Notkin (2013). Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering*, vol. 39, no. 10, October 2013, pp. 1358–1375.

Collins-Sussman, Ben; Brian W. Fitzpatrick; and Michael Pilato (2004). *Version Control with Subversion.* Sebastopol: O'Reilly & Associates.

Darcs (2003) Darcs. *Distributed. Interactive. Smart.* http://darcs.net. Accessed 19 March 2018.

Dewan, Prasun; and Rajesh Hegde (2007) Semi-synchronous conflict detection and resolution in asynchronous software development. In Bannon L. J., Wagner I., Gutwin C., Harper R. H. R., Schmidt K. (eds): *ECSCW'07: Proceedings of the 10th European Conference on Computer Supported Cooperative Work, 24 September 2007 – 28 September 2007, Limerick, Ireland.* London: Springer, pp. 159–178.

Dourish, Paul; and Victoria Bellotti (1992). Awareness and coordination in shared workspaces. In: *CSCW'92. Proceedings of the 1992 ACM Conference on Computer-supported Cooperative Work, Toronto, Ontario, Canada, 1 November 1992 – 4 November 1992.* New York: ACM, pp. 107–114.

EffectSizeCalculator (2017) Practical meta-analysis effect size calculator. https://campbellcollaboration.org/escalc/html/EffectSizeCalculator-SMD10.php. Accessed 19 March 2018.

German, Daniel M.; Bram Adams; and Ahmed E. Hassan (2015). Continuously mining distributed version control systems: an empirical study of how Linux uses Git. *Empirical Software Engineering*, vol. 21, no. 1, February 2016, pp. 260–299.

Git (2005) Git. *Fast version control system.* https://git-scm.com/. Accessed 19 March 2018.

GitChangesAnalyzer (2017) Python scripts for analyzing parallelism and conflicting changes in Git. https://github.com/coast-team/ParallelismAndConflictingChangesInGit. Accessed 19 March 2018.

GitHub (2008) GitHub. *Web-based Git repository hosting service.* https://github.com/. Accessed 19 March 2018.

Gousios, Georgios; Martin Pinzger; and Arie van Deursen (2014). An exploratory study of the pull-based software development model. In P. Jalote, L. Briand, and A. van der Hoek (eds): *ICSE'14. Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May 2014 – 7 June 2014.* New York: ACM, pp. 345–355.

Gutwin, Carl; Reagan Penner; and Kevin Schneider (2004). Group awareness in distributed software development. In: *CSCW'04. Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work, Chicago, Illinois, USA, 6 November 2004 – 10 November 2004.* New York: ACM, pp. 72–81.

Ignat, Claudia-Lavinia; and Gérald Oster (2008). Awareness of Concurrent Changes in Distributed Software Development. In Meersman R., Tari Z. (eds): *OTM 2008. On the Move to Meaningful Internet Systems, OTM*

Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008, Monterrey, Mexico, 9 November 2008 – 14 November 2008, Lecture Notes in Computer Science, vol 5331. Berlin, Heidelberg: Springer, pp.456–464.

IkiWiki (2015) Ikiwiki. https://ikiwiki.info/. Accessed 19 March 2018.

Kasi, Bakhtiar K.; and Anita Sarma (2013). Cassandra: Proactive conflict minimization through optimized task scheduling. In D. Notkin, B. H. C. Cheng, and K. Pohl (eds): *ICSE'13. Proceedings of the 35th International Conference on Software Engineering, San Francisco, CA, USA, 18 May 2013 – 26 May 2013*. Piscataway: IEEE, pp. 732–741.

LinuxKernel (2015) Linux kernel. https://www.kernel.org/. Accessed 19 March 2018.

LinuxKernelMailingList (2017) Linux kernel mailing list archive. https://lkml.org/lkml. Accessed 19 March 2018.

McKee, Shane; Nicholas Nelson; Anita Sarma; and Danny Dig (2017). Software practitioner perspectives on merge conflicts and resolutions. In: *ICSME'17. Proceedings of the IEEE International Conference on Software Maintenance and Evolution, Shanghai, China, 17 September 2017 – 22 September 2017*. Washington: IEEE Computer Society, pp. 467–478.

Mens, Tom (2002). A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering archive*, vol. 28, no. 5, May 2002, pp. 449–462.

Mercurial (2005) Mercurial. *Work easier, Work faster*. https://www.mercurial-scm.org/. Accessed 19 March 2018.

Perry, Dewayne E.; Harvey P. Siy; and Lawrence G. Votta (2001). Parallel changes in large-scale software development: an observational case study. *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 3, July 2001, pp. 308–337.

Reiher, Peter; John Heidemann; David Ratner; Greg Skinner; and Gerald J. Popek (1994). Resolving file conflicts in the Ficus file system. In: *USENIX'94. Proceedings of the USENIX Summer 1994 Technical Conference, Boston, Massachusetts, USA, 6 June 1994 – 10 June 1994*. Berkeley: USENIX Association, pp. 183–195.

Ruby (2015) Ruby on rails: The popular MVC framework for Ruby. http://rubyonrails.org/. Accessed 19 March 2018.

Samba (2015) Samba - opening windows to a wider world. https://www.samba.org/. Accessed 19 March 2018.

de Souza, Cleidson R. B.; David Redmiles; and Paul Dourish (2003). "Breaking the code", moving between private and public work in collaborative software development. In: *GROUP'03. Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work, Sanibel Island, Florida, USA, 9 November 2003 – 12 November 2013*. New York: ACM, pp. 105–114.

de Souza, Cleidson R. B.; David Redmiles; Li-Te Cheng; David Millen; and John Patterson (2004). Sometimes you need to see through walls: A field study of application programming interfaces. In: *CSCW'04. Proceedings of the 2004 ACM conference on Computer supported cooperative work, Chicago,*

Illinois, USA, 6 November 2004 – 10 November 2004. New York: ACM, pp. 63–71.

SpearmanRankCorrelation (2018) Spearman's rank correlation coefficient. https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient. Accessed 19 March 2018.

Yu, Weihai; Luc André; and Claudia-Lavinia Ignat (2015). A CRDT supporting selective undo for collaborative text editing. In A. Bessani and S. Bouchenak (Eds.): *DAIS'15. Proceedings of the 15th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems, Grenoble, France, 2 June – 4 June 2015*. New York: Springer-Verlag, vol. 9038, pp. 193–206.

Zimmermann, Thomas (2007). Mining workspace updates in CVS. In: *MSR'07. Proceedings of the Fourth International Workshop on Mining Software Repositories, Minneapolis, Minnesota, USA, 19 May 2007 – 20 May 2007*. Washington: IEEE Computer Society, pp. 1–11.