Real-time Collaborative Resolving of Merge Conflicts

Antti Nieminen
Tampere University of Technology
Korkeakoulunkatu 1, FI-33720 Tampere, Finland
antti.h.nieminen@tut.fi

***Abstract*—Merge conflicts are common especially in large and distributed software projects. There have been development to both make the conflicts less frequent by introducing better merge algorithms and to aid the user in resolving conflicts by better visualization and other means. However, one element lacking from the current conflict-resolving tools is the utilization of collective knowledge of project members, achievable through web-based collaboration. The person encountering the conflict may not have all the necessary information and understanding for resolving the conflict, in which case it would be beneficial to do the resolving collaboratively by all the parties who have been involved in causing the conflict or who otherwise have valuable information about the conflicted part of the project. In this paper, we present a process for real-time collaborative merging as well as a web-based tool supporting the process.**

***Index Terms*—real-time collaboration, version control**

I. Introduction

The World Wide Web — or simply the Web — has revolutionized collaboration. Tools such as social media and instant messaging help bring people around the world closer together. The Web has also improved collaboration in the field of software development. Some of the communication problems in software projects have been solved by web-based tools such as bug-reporting systems, wiki sites, blogs and other use of social media [13].

Despite the wide adoption of web-based tools to coordinate software projects, this trend has not really gone as far as to the actual programming. The code lines are still mostly written using similar code editors that have been used for decades. There have been some early attempts, such as [6], [4], to create a development environment where the process of writing code would be tightly integrated with web-based communication and collaboration tools. However, for such real-time collaborative environments to gain wider adaptation, they still need to answer a lot of unsolved questions related to issues such as version control, debugging, and testing in a collaborative environment [8].

Even though the real-time collaborative programming environments may not be ready to completely replace the traditional means of writing code, we see that such environments could even in the present state offer benefits in some use cases, more limited in scope and duration than a whole software project. Real-time collaboration is especially valuable in situations where there is a particular need for extensive cooperation and communication.

A prime example of a situation where there is a specific need for closer collaboration in a software project is the process of resolving merge conflicts. Merge conflicts occur when two lines of development are attempted to be merged into one but the version control system can not automatically execute the merge. In such a case, the conflict must be resolved manually by the project members. Typically the resolving is done by the person first encountering the conflict, possibly using some tool for visualizing the changes done by each party of the conflict. In some cases, though, it would be beneficial to resolve the conflict collaboratively by all the parties involved in causing the conflict.

In this paper, we describe a process for resolving a merge conflict in real-time collaboration. As a concrete artifact, we present a web-based tool supporting the said process for conflicts occurring in the Git[1] version control system.

II. Background

A. Real-time Collaboration

A collaborative real-time editor (CRE) enables multiple users to edit the same document concurrently and to see each others' edits in real time. The first CRE was introduced as early as 1968 in what was to be called "The Mother of All Demos"[2]. Lately, along with the Web becoming a viable application platform, CREs have gained a larger audience. For example, Google Docs[3] is a widely used platform utilizing real-time collaboration for creating documents.

Collaborative real-time editors must somehow manage concurrent edits by different users by applying each users' changes into the shared document as well as possible. A naive approach of, for example, just inserting text in a given position P in the shared document does not work properly because after concurrent edits by other users, P often does not refer to the position the user intended. Two of the most often used methods for dealing with the concurrency and synchronization issues are Operational Transformation [14] and Differential Synchronization [3].

Our solution for collaborative conflict-resolving is based on CoRED [6][4], which is is a web-based CRE for software de-

[1] http://git-scm.com/
[2] http://en.wikipedia.org/wiki/The_Mother_of_All_Demos
[3] http://docs.google.com
[4] http://cored.cs.tut.fi

COLLABORATECOM 2012, October 14-17, Pittsburgh, United States

DOI 10.4108/icst.collaboratecom.2012.250435

velopment. It is implemented as a Vaadin [5] component containing both client-side and server-side functionality. CoRED uses Ace[5], an open-source code editor component written in JavaScript, for basic code editor features. Ace offers syntax highlighting, automatic indenting and other helpful features for various programming languages.

CoRED utilizes Differential Synchronization algorithm (described in [3]) to handle concurrent edits. Along with plain text, CoRED documents may also contain *markers*. Markers are sections within a text document that retain their logical positions. They have a start position and an end position, and they are used for example to present code errors or user notes within a document. In the case of concurrent edits, the position of the markers, along with the textual edits, is adjusted in accordance with the Differential Synchronization algorithm.

B. Version Control

Version control systems (VCSs) [12], [10] are tools for managing a collection of documents, typically a software project. VCSs enable users to inspect the history of the project and make it possible to revert back to an earlier version. All the modern VCSs allow for a project to have multiple independent lines of development, called *branches*. For example, a project may have a stable release branch and another branch for development. Thus, new features are first committed to the development branch, from where some or all of the modifications may later be *merged* to the release branch.

The purpose of *merging* is to combine two versions of a file into one in such a way that the semantics of both files are preserved. Although finding a merge that preserves the semantics of both sides of the merge is in a general case an algorithmically undecidable problem [1], automatic merge algorithms are very useful in practice. When a merge algorithm can not execute the merge automatically, a *merge conflict* is risen, and the conflict must be resolved manually.

There are a lot of tools to help the user to resolve a merge conflict. They often offer some kind of visualizations of the changes done by each party of the conflict, and possibly some other aids such as shortcuts for easily choosing either one of the conflicted sections. Some of the existing merge tools are KDiff3[6], WinMerge[7], Diffuse[8], and SourceGear DiffMerge[9]. None of the existing merge tools we know of support collaboration of multiple users.

Currently, one of the most popular version control systems — and also the one our implementation can currently be integrated with — is Git. Git is a distributed VCS, designed to be efficient and to support distributed, non-linear development. Branches in Git are lightweight, encouraging developers to utilize them heavily and merge frequently.

[5]http://ace.ajax.org/
[6]http://kdiff3.sourceforge.net/
[7]http://winmerge.org/
[8]http://diffuse.sourceforge.net/
[9]http://www.sourcegear.com/diffmerge/index.html

Git offers support for integrating a *mergetool* to a conflict-resolving process. Git can give the merge tool four files as arguments: `LOCAL` contains the locally edited version of the conflicted file, `REMOTE` is the version to merge with, `BASE` is their common ancestor, and `MERGED` is the combination of `LOCAL` and `MERGED` with the conflicted areas marked. `MERGED` is also the working file where the mergetool should write the result of the merge.

III. The Process of Collaborative Resolving of Merge Conflicts

When merging, the version control system can perform the merge automatically in the majority of the cases (90% according to one study [11]). For the remaining cases, user input is needed to resolve the conflict. Some of the conflicts are easily resolved by the person first encountering the conflict. Occasionally, though, the conflict is so difficult to resolve that the user can not do it by himself in such a way that the changes done in both of the merged versions are taken into account, and without causing any additional errors. The difficult conflicts are situations where the resolver does not fully understand the changes done by the other party, based on the information he has, such as the source code and the commit messages.

A. Basic Workflow

To resolve a difficult conflict, it would be helpful to have input by all the parties that have contributed to the conflict. Consider an example where Alice developed a feature A while Bob concurrently developed a feature B. After both finished, Alice tried to merge her changes with those of Bob. The features A and B overlapped each other somehow, thus the attempt to merge resulted in a conflict. In this case, it would be beneficial if both Alice and Bob could participate in the conflict resolution process. In some cases, it would be useful to even involve another developer, Chuck, who have been working in the same part of the project earlier.

Our solution provides a way for a real-time collaborative conflict resolution. When Alice, using the Git version control system, encounters a conflict she can not resolve by herself, she can start up a new conflict-resolution session by launching our tool. The tool uploads the conflicted file to a server, and launches a web browser pointing to a URL (Uniform Resource Locator) that opens a conflict-resolving session. In the session, Alice sees the conflicted file with the conflict areas marked. She can then edit the file in the browser, mark the conflicted sections as resolved and finally choose the resolved version to be downloaded back to her local file system to be committed. That is how it would be done if Alice can resolve the merge just by herself. The real benefit of our tool, however, is the possibility to invite others to the same conflict-resolution session, which will be described next.

When Alice, the original resolver, has opened the conflicted file in the web-based merge tool, she has the option to invite other developers to collaboratively resolve the conflict. Inviting is done by providing others a URL pointing to the same conflict-resolution session. The URL can be given to anybody

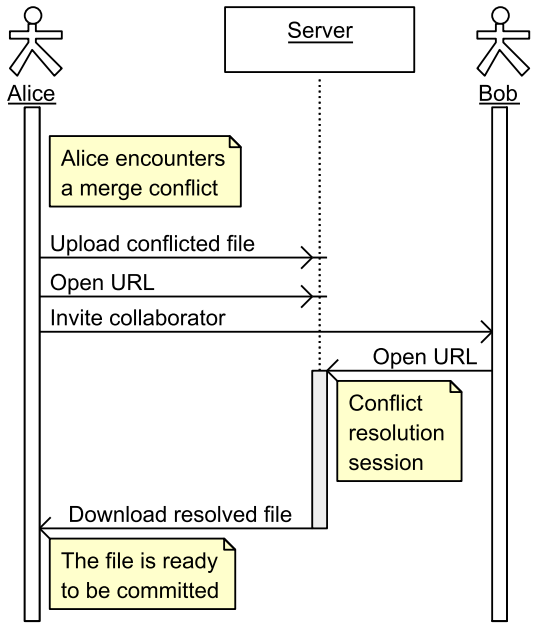

Fig. 1. The basic workflow of collaborative conflict-resolving.

by any means, such as pasting it into a instant-messaging chat with Bob. The name and email address of the author of the commit that the user is trying to merge with is provided for convenience. Our tool also contains a rudimentary Doodle[10] integration. Doodle allows people to schedule an event (such as a conflict resolution session) by letting each participant mark the time slots on which they are available, and thus choose the best matching time for the event. Similar integration could be done to any collaboration tool. In some cases, a commit is combined with continuous integration (CI) system, and then CI could initialize the collaborative resolving.

Having connected to the merge tool, all the collaborators can edit the conflicted file in real time and mark conflicted sections as resolved. The tool also offers a chat box where the users can discuss issues such as how to best resolve the conflict. When all the conflicted sections are marked as resolved, the original resolver can apply the merge. When the merge is applied, the conflict resolving session ends, the tool downloads the newly resolved version of the code back to the original resolvers local file system and notifies Git of a successful merge. Thus, the file is ready to be committed. A sequence diagram of a successful merge resolution session is presented in Figure 1.

The URL provided to other collaborators, and thus their view of the tool, is a bit different from that of the original resolver. Only the original resolver has the option to either cancel the merge or to apply the merge. Also, the possibility to schedule a Doodle event and other interface elements for inviting collaborators are only available for the original resolver. The original resolver is also the only person that needs to have our tool configured as a Git mergetool. Starting the tool is like starting any traditional single-user Git mergetool, except that unlike usually, our tool launches the conflict-resolving session inside a web browser. Other participants do not need to have anything installed except a web browser with which to connect to a URL they received from the original resolver.

B. Delayed Resolving of a Conflict

When a need for a collaborative conflict-resolving emerges, other collaborators may not be available at that very moment. In such a case, the original resolver may choose to discard the merge for the moment and wait for all the collaborators to schedule a time for a conflict-resolution session. The original resolver may continue to work on her own branch and later incorporate all her latest changes to the same conflict resolving session that has already been scheduled.

Let us say that Alice tried to merge her commit A_1 with Bob's concurrent commit B_1, resulting in a merge conflict M_1. Alice could not resolve the conflict by herself, so she invited Bob to a collaborative conflict-resolving session. Unfortunately, Bob was unavailable at that time, so Alice, after sending an invite to Bob, chose to discard the merge for now and to continue further development on her own branch while waiting for Bob to schedule a conflict-resolving session. Later, before the session takes place, Alice can attempt to merge the latest commit in her branch with those of Bob, resulting in a conflict M_2. Even though the two conflicts, M_1 and M_2, are different from the version control systems point of view, the same conflict-resolving session and the same URLs can be used without the need to do another round of inviting and scheduling. In the unlikely event of the merge between A_2 and B_1 executing automatically, the whole collaborative conflict-resolving session is canceled.

IV. Implementation

Our implementation of a tool for real-time collaborative conflict-resolving consists of two components: a Git mergetool that integrates our solution to a Git version control system and the actual user interface of our tool, a web application for collaborative conflict resolution.

The first component of our tool is a locally executed script that implements the Git mergetool interface. Namely, it takes as an argument a list of filenames as described earlier, and returns zero indicating a successful merge and nonzero for an unsuccessful merge. The mergetool is written in Python[11] so it can be run on all the major operating systems. The tool collects some information from Git, such as the authors of the current and the merged commits, and sends them, along with the conflicted file, to the web application via HTTP (Hypertext Transfer Protocol).

After successfully sending the file, the mergetool launches a web browser pointing to a URL that it received from the web application. Then, the mergetool makes another HTTP

[10]http://doodle.com/

[11]http://www.python.org/

connection to the web application to wait for the original resolver to either declare the merge as successful, or to cancel the merge, after which the tool returns the result back to Git.

The implementation of our web-based conflict resolution tool is based on CoRED, enabling real-time collaborative editing of the conflicted file. The conflicted parts of the file are presented as CoRED markers. The current implementation directly gets the conflicted sections from the `MERGED` file provided by Git. In addition to editing the file manually, users are offered a shortcut to use any of the two versions of a conflicted part. Users can also mark a conflicted section as resolved and chat with each other. All the changes are sent to other collaborators nearly instantly, using HTML5 Web Sockets [7] if available.

When creating a new conflict-resolving session, the web application generates an authentication token. The token is part of the URL sent to the mergetool script. The authentication token is a randomly generated string, unique for each $(Session, OrigResolver)$ pair, where $Session$ is the session id and $OrigResolver$ is a boolean indicating whether the user is the original resolver. The authentication token (as well as the URL containing it) acts as a password: it is only possible to join a conflict-resolving session if you know the URL.

V. Discussion

In this paper, we described a novel approach to resolving merge conflicts: utilization of real-time collaboration among software project participants. As a concrete contribution, we presented a web-based tool that enables multiple users to collaboratively resolve merge conflicts, occurring in a Git version control system.

Our tool, although implemented for Git, could also be used with other version control systems with modest modification. Only the local script (mergetool in the case of Git) needs to be reimplemented and the web application can be used as is. In fact, the web application could be used to resolve any kind of merge conflict; it does not even have to originate from a version control system. Just a local executable to upload and download data to/from web application is needed.

The web-based interface could be further improved by implementing more of the features seen in existing single-user merge tools, such as side-by-side views of both versions and their common ancestor, better visualization of changes, and more intuitive methods for choosing which part of which version to use. Additionally, the presence awareness [2] of the collaborative tool could be improved by, for example, showing the cursor positions of other collaborations, and clearly presenting which part of the code was written by whom, and whether it was written during the conflict-resolving session or earlier. Another important point for improvement would be to provide a more comprehensive view of the conflicted project, not just a single conflicted file as in the current implementation. However, in terms of Git integration, a problem with this multi-file approach is that resolving conflicts spanning multiple files is not directly supported by the Git mergetool workflow.

Another interesting question is who should be invited to a conflict-resolving session. An obvious choice, along with the original resolver, is the author of the commit that is on the other side of the merge. However, there might be cases where additional collaborators provide benefit. In version control systems, it is possible to find out who have been editing a certain line in a file, and use that information to invite people who have been editing the conflicted file lately. Even more advanced methods for finding a person with the most expertise in some aspect of a software project could be used, such as presented in [9].

As future work, we aim to test our solution for merge conflict resolution in actual real-world cases, first in a limited setting, and hopefully later, as the tool is released as open-source software, with a larger number of users. Based on those results we may better be able to justify our approach as well as understand the situations where it provides benefits. After real-word testing it is also easier to see how the tool can be further improved.

References

[1] V. Berzins. On merging software extensions. *Acta Informatica*, 23:607–619, 1986. 10.1007/BF00264309.

[2] J. Espinosa, S. Slaughter, R. Kraut, and J. Herbsleb. Team knowledge and coordination in geographically distributed software development. *J. Manage. Inf. Syst.*, 24(1):135–169, July 2007.

[3] N. Fraser. Differential synchronization. In *Proceedings of the 9th ACM symposium on Document engineering*, DocEng '09, pages 13–20, New York, NY, USA, 2009. ACM.

[4] M. Goldman, G. Little, and R. C. Miller. Real-time collaborative coding in a web ide. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, UIST '11, pages 155–164, New York, NY, USA, 2011. ACM.

[5] M. Grönroos. *Book of Vaadin, 4th ed.* Uniprint, Turku, Finland, 2011.

[6] J. Lautamäki, A. Nieminen, J. Koskinen, T. Aho, T. Mikkonen, and M. Englund. Cored: browser-based collaborative real-time editor for java web applications. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, CSCW '12, pages 1307–1316, New York, NY, USA, 2012. ACM.

[7] P. Lubbers and F. Greko. Html5 web sockets: A quantum leap in scalability for the web. http://www.websocket.org/quantum.html. [Online; accessed 2012-09-17].

[8] T. Mikkonen and A. Nieminen. Elements for a cloud-based development environment: online collaboration, revision control, and continuous integration. In *Proceedings of the WICSA/ECSA 2012 Companion Volume*, WICSA/ECSA '12, pages 14–20, New York, NY, USA, 2012. ACM.

[9] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 503–512, New York, NY, USA, 2002. ACM.

[10] B. O'Sullivan. Making sense of revision-control systems. *Commun. ACM*, 52(9):56–62, Sept. 2009.

[11] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, July 2001.

[12] D. Spinellis. Version control systems. *Software, IEEE*, 22(5):108 – 109, sept.-oct. 2005.

[13] M.-A. Storey, C. Treude, A. van Deursen, and L.-T. Cheng. The impact of social media on software engineering practices and tools. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 359–364, New York, NY, USA, 2010. ACM.

[14] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, CSCW '98, pages 59–68, New York, NY, USA, 1998. ACM.