Concurrent Coding with the Help of Live Sharing Tools

Adrian Vinojcic
T1, Software Repositories|Collaborative Engineering
11904250/521
adrian.vinojcic@jku.at

Alexander Voglsperger
T1, Software Repositories|Collaborative Engineering
12005568/521
alexander.voglsperger@jku.at

***Abstract*—When working in teams, simultaneously working on projects creates the problem that merging the different versions of code creates complicated extra steps. This is a pity, because working on a project together has many positive side effects, which can make it more pleasurable to work and boost the productivity of a programmer. Using *Live Sharing Tools*, programmers can work on one project in near real-time on different computers from anywhere in the world. This will remove the need to merge versions which programmers create, with the benefit of seeing what their colleagues are currently working on.**

***Index Terms*—concurrent, live, collaborative, coding**

I. Introduction

Writing software is a task which in nearly all cases cannot be handled alone, since projects tend to get ambitious very quickly. Therefore, programmers often work together to create reliable software. This means that programmers have to read the same files and also edit them. This is the problem discussed in this article. This process can be done very inefficiently by shift work, where person *A* works, then person *B* and so on. The better solution is to work on the project concurrently, which is most common these days. Now the problem arises, that the work has to be split up somehow.

Doing so has multiple solutions:

- Splitting up the workload strictly
- Using a Source Control Manager (SCM)
- Using a Live Sharing Tool (LST)

Every solution has its own benefits and drawbacks. In this chapter we will provide a short explanation of the listed concepts, then we will explain each idea in detail in their dedicated chapters.

A. Labor division

Splitting up the project between the developers creates an isolation. Thus, every individual can write their part of the program in their respective Integrated Development Environment (IDE) on their own machine and test it without the interference of others. If the division of the tasks is done cleverly and everyone delivers a good Application Programming Interface (API) this can go smoothly. Nevertheless, this does not happen too often, since requirements change and not everything can be planned in advance. As soon as all the programmers are finished with their part, the code has to be merged somehow. Most likely, programmers will need to adapt their code in a certain way in order to be compatible with the code of their co-workers. This often results in an expensive phase, since everyone needs to figure out how the code of the others is structured and how it works. Furthermore, the process of splitting up the work very strictly is a time-consuming process as well. These precious resources can be better utilized in other, more effective, ways.

B. SCM / Repositories

A more suitable method would be to upload the changed code to a central platform regularly and download it when picking up the work again. On such a platform other developers are able to get the latest version of the project, adapt to the changes someone else made and proceed working on the most recent version or even a different branch.

As of now, this is the most commonly practiced method of collaboration. This way of handling the issue is most commonly done using *Git* and will be elaborated further in Chapter II. Clearly the advantages are the existence of a central storage- and versioning system of some kind. There is obviously a significant improvement over merely exchanging the files via disk or a network share. Disadvantages of using SCM are merging new code to the source when one contributor introduced incompatible code to another ones. This process is called *manual merging* and is very tedious, since the developer has to pull the new source and implement the new features again. *Automatic merging* in contrast is when there are no missing commits between the versions and the SCM can merge the versions itself [12].

C. Live Sharing

While software repositories give a lot of benefits, our suggestion is to extend them by using LST in addition. A LST at its core is mostly a hosted text document on a server [8], which could be done by a SCM as well, to which every participant has access to. This server can be something as simple as a file system which multiple people can write to, as described in chapter III-A. It can also have a lot of features implemented like syntax checking, which only propagates text when the syntax is not malformed by the change, as it is done by [5] with *Collabode*. The workings, challenges, and compromises of LST are explained in detail in chapter III.

II. Current state of collaborative coding

The most common way of collaborative coding is using Git. This tool allows programmers to have their own version locally and one unified version on a server for all contributors. Local changes can be merged with the global version, which then can be fetched from others [3]. As an example, according to their latest report, *GitHub* was used by over 73 million developers in 2021 [4].

This method already allows developers to cooperate more while working on a project, but there still exists a separation where one can not see the other's changes on the project until changes are pushed and then also pulled. Meaning that merging still requires a great effort in most cases, especially if users do not push and pull at a high frequency.

Another common practice is *pair-programming* (also in combination with SCM), where two programmers sit in front of one machine doing their work in a single IDE. They can exchange their thoughts and reduce the complexity of the written code. This practice also reduces their susceptibility to errors, since two pairs of eyes are more likely to spot mistakes. This in return reduces the cost of the project overall by detecting errors earlier in the process [1]. Especially the effectiveness of pair-programming could be increased when multiple people would be able to work on a project simultaneously while the changes are being propagated to the colleagues in real time.

III. Live Coding

In order to share code and work on it concurrently, one has to first implement a tool to do so. In this chapter, we are discussing requirements for this challenge and possible solutions to this problem.

A. Plain Text Sharing

Before one can code, the problem has to be reduced to sharing simple plain text, later additional features like testing and debugging can be added upon the existing framework.

Simple solutions allow working on multiple files, with the drawback being that a file cannot be edited simultaneously. This means that tasks still have to be split up strictly, however everyone can see each other's changes in real-time. It is also possible to apply a smaller granularity and lock only sections or lines, to increase the level of collaboration, as described in [7].

The next idea is to share one file for editing simultaneously. This method is known by the term *teletyping*, which only consists of real-time text editing without any other features. Such a system is mostly characterized by three aspects [7]:

- *Real Time*: the time to response should be minimized
- *Distributed*: users should be allowed to be in different locations (i. e. different networks) and work on the document
- *Unconstrained*: it is not important, when a user edits the document

A framework which provides such a reactive, concurrent real-time editing environment is the Operational Transformation (OT) framework [8]. It deals with basic the consistency of the written text, but ignores anything context specific, since it is not made for that. The basic concept is that every client has an own version of the document. To edit the document the framework first writes every keystroke to the own version of the document and then sends a corresponding operation in the form of, `insert/delete[pos, char]` where `pos` is the position in the text and `char` is the inserted/deleted character, to the other client(s) [9].

If a concurrent modification happens, the framework adapts some operations, if needed, to keep it consistent with the order of operations already transmitted and received as shown in Figure 1.

A running example of this consistency protection would be the following:

- The document initially consists of the text `'abc'`
- Editor A writes `delete[0, 'a']`
- Editor B writes `insert[3, 'd']`
- Tool propagates change from A `delete[0, 'a']`
- Tool propagates change from B `insert[3, 'd']` but with adopted values `insert[2, 'd']`
- In the end everyone has the same version of the file consisting of the test `'bcd'`

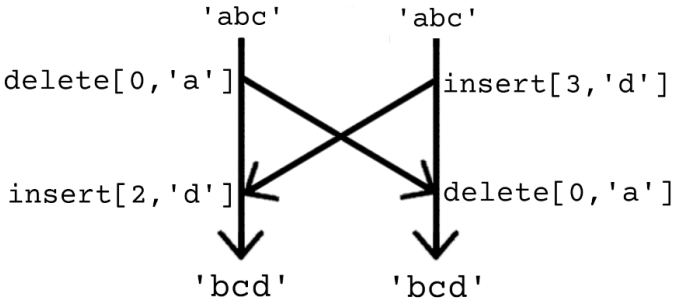

Fig. 1. A running example of the OT

Advanced tools feature a type of collaborative awareness, where it feels like sharing an IDE with multiple programmers. This ranges from seeing each other's cursor with a name to collaboratively debugging code, where everyone can see what is happening at each debug step [5].

After reading about a lightweight categorization, we are going to take a look at the problems that we have to face when trying to create a LST.

B. Maintaining Synchronization

Maintaining synchronization between multiple clients is the main challenge, since all other features build upon this. The usually underlying algorithm of a distributed OT would break if clients diverged by more than one symbol [8]. OT deals with generic consistency, but doesn't take context specific challenges into account. A solution to such problems are centralized architectures, which establishes a *client-to-server*-communication instead of a *client-to-client*-communication.

Therefore, the client sends valid changes to the server, which then saves and distributes them to all other clients [8].

C. Naive synchronization of incomplete/broken code

Here, we want to take a look at problems which may occur even when synchronization is working as intended. When going back to using a type of teletyping, where each character gets naively synchronized without any context, transmission of unfinished code is likely going to happen, which would cause a flood of errors by the compiler or IDE. It is incredibly frustrating for a programmer to work on a project to enter a buildbreaking state just because another contributor changes something in the code somewhere else. It may also halt other users from working, even though they didn't contribute to breaking the code in the first place.

D. Multiple States

Goldman, Little, Miller [5] and also Levin and Yehudai [8] both describe models in which they tackle this issue in various ways. As an example, both groups mention a type of state-driven synchronization where the code is only exchanged when it can compile without errors. This means that each programmer has a local copy on their machine, which they work on. All local copies gets compiled or at least syntactically checked before they are sent to the server. When the code in one user's file is in a *buildable/compilable* state, it is getting distributed to all other clients, as we can see in Figure 2. Otherwise, the broken code is withheld until a working state is reached again. This makes it possible, that others can keep working in a buildable state without the interference of the broken code part. This means, that when one enters a buildable state again, the version controlling algorithm commits the changed code to the public version which is then in turn getting distributed to all clients working on the project. This makes it possible to isolate individual programmers to a certain degree while also keeping the global version as up-to-date as possible without breaking the other programmers' code.

The isolation thought comes from the *commit* to a SCM, where they are expected to commit to a certain quality[8]. Quality in the sense of working on locally tested code to ensure a reduced amount of bugs in the public version. The main disadvantage with this solution is, that the longer a programmer stays in an unbuildable state, the more the version may drift apart in respect to the actual current public state. Should a conflict be detected, the programmers are informed, and it is up to them how to resolve the conflict and thus the final state of the code [5][8].

Multiple states may also require some automatic combining, if the states are diverged too much from the public state. For this use case, language aware tools are a good fit. They often come with different types for detecting changes and handling them. E. g. when moving blocks of code or refactoring files. They allow combining code that seems quite different, but originated from the same source [11].

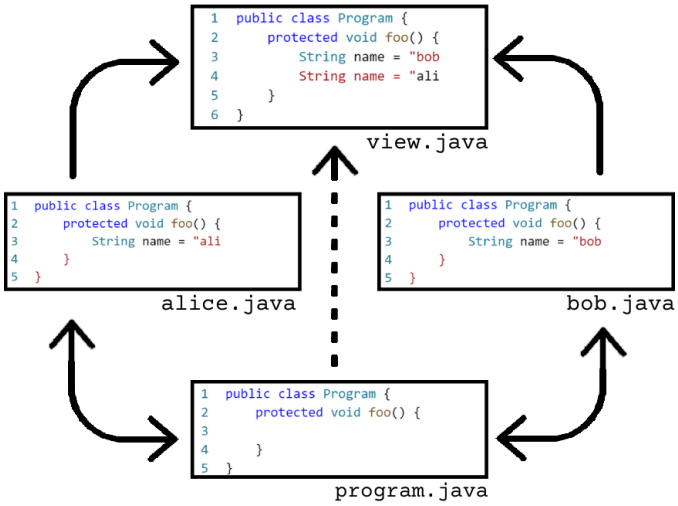

Fig. 2. Scheme of having multiple States

E. File Locking

Another possibility is file locking, where we mainly differentiate between *pessimistic* and *optimistic* locking. For pessimistic locking, on the one hand, only one editor can make changes to a file at a time. Co-editors only have read access to the file in the meantime. If they want to edit something, they have to wait until the editor owning the lock releases it again. This ensures data integrity but may harm productivity in the way that dependencies of, for example, refactored functions are not considered in other files. Concurrent activity may be reduced depending on the structure of the project. On the other hand, optimistic locking allows concurrent editing as much as possible, but disallows programmers to edit the same part of code at the same time. The scope of a method has proven to be a good granularity for locking[8] parts of code. This is better suitable for code where conflicts are rare. However, detecting such conflicting changes and how to treat them is a bit of a problem.

Since optimistic conflicts would likely result in manual merges, Levin and Yehundai believe that a pessimistic approach would be better suited. This is under the assumption that developers rather wait a reasonable time than manually merge possible conflicting versions. There are some hybrid models too, which lock dependencies as well. This approach requires some sort of language understanding when locking parts of code and the respective relationships in other parts. If multiple children with a common parent are edited, the OT-framework will be used. In general, the system has to decide what depends on which element as soon as possible before locking a file partially or completely. Not doing so may result in other programmers using/changing dependencies since they are not locked yet, which in return would result in breaking changes again [8].

F. Comparison of LST solutions

The solutions, we are going to take a look at, are *Microsoft's* Visual Studio Live Share (VSLS) [10], *JetBrains'* Code With Me (CWM) [6] and *Genuitec's* CodeTogether (CT) [2]. All of these offer a free version, but CWM and CT also offer paid variants, which contain additional features. Each tool offers *end-to-end-encryption* in their free version and do not transfer the entire workspace to non-host machines, but instead fetch data on demand. VSLS is capable of utilizing a Peer-to-Peer (P2P) connection if possible and offer *relay-servers* if P2P is not possible due to network constrains like Network Address Translation (NAT) or firewall. CWM and CT require a lobby-server, which can be hosted on-premise in the own infrastructure.

Another useful feature is the use of a browser-based editor instead of a fully fledged IDE. This feature is offered by VSLS and CT. Even though CWM does not offer a browser-editor at the time of writing, it offers a script that can be run to join a collaborative session without the need to install a full IDE. In contrast to VSLS and CWM which only work in their corresponding software suite, CT can be used in *VS Code*, *IntelliJ* and *Eclipse*. Furthermore, CT also allows user from different IDE's to work together.

However they are not perfect and still have long a way to work flawlessly. As an example, we faced a problem while writing this article with VSLS, where files were too large. Recompiling the LaTeXproject, the whole *PDF*-file changed, which required synchronization of a bigger file. But instead of synchronizing that file, it resulted in not synchronizing the whole project correctly anymore. Another problem that occurred were somewhat frequent disconnects, which required a restart of the connection and sometimes even restarting the session itself.

Each LST comes with its own benefits and drawbacks, nevertheless they accomplish the main task of relatively easy *live collaborative coding* and removing the need to merge conflicting versions of code.

IV. Conclusion

A. Discussion

In this article, we introduced the problem programmers face when working concurrently on a project and how to work collaboratively. We briefly explained different methods such as labor division I-A, SCM I-B and live-share I-C. We took a look at the current state of collaborative coding II where we explained which problems we currently face. In live coding III we gave some examples for the requirements and the challenges that we have to overcome for a good, reliable live coding experience. Last but not least, we compared some of the more popular solutions III-F in their key aspects. As personal preference, we think *Microsoft Visual Studio Code* with the *Live Share*-plugin is a pretty solid solution, since it supports almost any language and offers many features.

B. Further Work

As of now, we are at a point where collaborative work is easily done, with many tools to aid us. Nevertheless, development should never stop. From our point of view, development should go into the direction of LST a lot more. Our proposal is to further develop SCM to include LST out of the box. Furthermore, there is still a lot of room for improvement on the experience of LST such as better synchronization and handling of larger progress in files at once like explained in III-F.

References

[1] Alistair Cockburn and Laurie Williams. "The Costs and Benefits of Pair Programming". In: Addison-Wesley, 2000, pp. 223–247.

[2] Genuitec. *CodeTogether*. https://www.codetogether.com. Accessed: 2022-04-20.

[3] Git-SCM. *About – Git*. http://git-scm.com/about. Accessed: 2022-04-18.

[4] GitHub. *The State of the Octoverse*. https://octoverse.github.com. Accessed: 2022-04-18.

[5] Max Goldman, Greg Little, and Robert C. Miller. "Real-time collaborative coding in a web IDE". In: *MIT web domain* (Oct. 2011). URL: https://dspace.mit.edu/handle/1721.1/72493 (visited on 03/23/2022).

[6] JetBrains. *Code With Me*. https://www.jetbrains.com/help/idea/code-with-me.html. Accessed: 2022-04-29.

[7] Bo Jiang, Jiajun Bu, and Chun Chen. "Preserving Consistency in Distributed Embedded Collaborative Editing Systems". In: *Embedded Software and Systems*. Springer, Berlin, Heidelberg, 2005. DOI: 10.1007/11535409_88. URL: https://link-1springer-1com-1007e96r600b5.han.ubl.jku.at/chapter/10.1007/11535409_88 (visited on 04/27/2022).

[8] Stanislav Levin and Amiram Yehudai. "Collaborative Real Time Coding or How to Avoid the Dreaded Merge". In: *arXiv:1504.06741 [cs]* (Apr. 2015). arXiv: 1504.06741. URL: http://arxiv.org/abs/1504.06741 (visited on 03/23/2022).

[9] Yang Liu et al. "Formal Verification of Operational Transformation". In: *FM 2014: Formal Methods*. Ed. by Cliff Jones, Pekka Pihlajasaari, and Jun Sun. Cham: Springer International Publishing, 2014, pp. 432–448. ISBN: 978-3-319-06410-9.

[10] Microsoft. *Visual Studio Live Share*. https://docs.microsoft.com/en-us/visualstudio/liveshare/. Accessed: 2022-04-29.

[11] Santos Pablo. *Put your hands on a programming-language-aware, refactor ready, merge tool*. https://blog.plasticscm.com/2013/04/put-your-hands-on-programming-language.html. Accessed: 2022-05-26.

[12] Xiaoqian Xing and Katsuhisa Maruyama. "Automatic Software Merging using Automated Program Repair". In: *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*. 2019. DOI: 10.1109/IBF.2019.8665493.