Peer to Peer collaborative editing based on a git repository

BACHELOR THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Stefan Gussner

Registration Number 01527253

elaborated at the
Institute of Information Systems Engineering
Research Group for Industrial Software
to the Faculty of Informatics
at TU Wien

Advisor: Thomas Grechenig
Assistance: Johann Grabner

Vienna, September 3, 2019

Kurzfassung

Echtzeitkollaborationssoftware für Softwareentwicklung ist für den Einsatz in Unternehmen nicht uneingeschränkt verwendbar. Aktuelle Werkzeuge, wie Atom Teletype, sind keine Erweiterung eines Versionskontrollsystems, sondern versuchen Versionskontrollsysteme zu ersetzen, oder sind nicht für kontinuierliche Sitzungen ausgelegt. Dadurch gehen einige nützliche Features von Versionskontrollsystemen, wie Branches und das Ignorieren von Dateien, verloren. Auch die Nachvollziehbarkeit der Änderungen ist nur eingeschränkt gegeben.

Es wurde eine Erweiterung für Visual Studio Code entwickelt, die Peer to Peer Echtzeitkollaboration ermöglicht und Änderungen auf Git-Branches aufbaut, anstatt einfach Dateien zu synchronisieren. Echtzeitänderungen sind nur innerhalb eines Branches sichtbar und Dateien, die durch das Versionskontrollsystem ignoriert werden, werden auch von der Erweiterung ignoriert. Um Nachvollziehbarkeit zu gewährleisten, werden Echtzeitänderungen in Git committet. Änderungen können nach Autorin oder Autor committet werden, um nachvollziehen zu können, wer eine Änderung vorgenommen hat. Die Erweiterung wurde gegenüber aktuellen Lösungen evaluiert und ist die einzige Echtzeitkollaborationssoftware, die Git-Branches bei der Synchronisierung von Änderungen beachtet.

Schlüsselwörter

Echtzeitkollaboration, Git, Peer to Peer, Softwareentwicklung.

Abstract

Real-time collaboration tools for software development do not cover use cases common in corporate environments. Current tools do not offer an extension to a version control system but are trying to substitute a version control system or are not designed for long collaboration sessions. This limitation leads to the loss of useful features of the version control system like branches and ignoring specific files. Furthermore, the accountability for changes is reduced or lost.

An extension for Visual Studio Code has been developed, which enables Git branch-based peer to peer real-time collaboration to address these limitations. Real-time changes are only visible to other peers on the same Git branch. Files ignored by Git are ignored by the extension as well. For ensuring accountability, changes are committed to Git by their author. The extension has been validated against state-of-the-art tools and is the only currently known real-time collaboration tool respecting Git branches when synchronizing changes.

Keywords

Contents

List of Figures

List of Tables

List of Listings

1   Introduction

Millions of developers are collaborating[1] on source code every day. Especially in corporate environments, tools for collaboration on source code, such as Git[2], are an integral part of the workflow. Those tools are not just for managing versions and enabling merging conflicts but are vital for communication.

1.1   Problem Description

When multiple people are working on solving a problem, it is not clear who is implementing which part. Current version control systems (VCS) conflict resolutions require manual conflict resolution if multiple people have modified the same file concurrently. For a group to discuss a problem, all the code has to be committed and pulled by everyone first. This step introduces friction. To solve these and other problems, real-time collaborative editors have been created. However, current implementations of real-time collaboration editors or plugins are unaware of the underlying version control system. Current real-time collaborative editors are based on the idea of just sharing files of a host machine or a single source of truth file on a server. None of the current solutions are using edit histories based on versions of files known to the version control system.

In state-of-the-art collaborative code editors, it is not possible for a user to easily see who made specific changes. Usually, all the changes are bundled in one commit, and the accountability is lost. To convert this concurrent model to a commit understood by Git, it should be possible to stage changes by the author. This approach might require more sophisticated logic than just interpreting changes as strings. A command could be modified by two users and applying only half the changes as part of a commit could result in invalid syntax or semantic. Interpreting changes as an abstract syntax tree or in other formats could reduce conflicts introduced by modifying the same command on different mashines and then committing only the changes of one author.

1.2   Expected Results

- Real-time collaboration on source code should be based on a Git project by continuous tracking of code changes.

- Aiming to decrease the overhead of splitting tasks by enabling everyone in the team to see what other people are working on.

- Allowing discussions about source code that has not yet been committed.

By building a prototype it should be possible to evaluate the impact of Git-based real-time collaboration on common workflows in corporate environments. Changes should be committable by each author. To be able to compile source code, changes of other people can be toggled off. Using Git as a base enables opportunistic real-time collaboration. In other words, if a connection is possible, changes are propagated to other people working on the same branch. If not, the changes are sent

---

[1]   https://octoverse.github.com
[2]   https://git-scm.com/

when a connection is available [26]. Therefore, it should be analogue to the benefits of moving to a decentralized VCS, as described in [1]. This is because, as in Git, disconnecting from the network will not block working on a shared document.

1.3 Motivation

Current implementations of real-time collaboration tools are not designed with version control systems in mind. There is usually no way to transform changes into the version control system without losing information about the author of changes. Furthermore, establishing a session is uaually unnecessarily expensive because the initial state of documents is tranfmitted to all other endpoints despite the endpoints usually already having a (maybe slightly outdated) copy of the document stored.

Transforming real-time collaborative edits into regular Git commits by author reduces much friction in the adoption of real-time collaboration software. This is because the semantic of a commit is very similar to a regular Git-based workflow. When working with current implementations for real-time collaboration, changes can only be imported into Git by committing entire files or entire changesets without much regard for grouping changes into commits. Using a peer to peer solution with the ability to deal with disconnect events using information already known to the version control system drastically increases the ease of use. Ideally, a user will not even have to think about using the extension.

2   Fundamentals

In this section, multiple solutions for real-time collaboration are analysed. The results from the literature survey are used as a basis for the requirements analysis.

2.1   State of the Art

This section describes and discusses the current state-of-the-art tools for real-time collaboration.

2.1.1   Available Software Tools

In this section, currently available software solutions for real-time collaboration are discussed.

Teletype for Atom

Teletype for Atom[1] is a project enabling editing files peer to peer. It is based on [14], [15] and [4]. With Teletype, it is possible to edit files currently opened by the "host". The files are only persisted on the "host" not on all peers[2]. Therefore, disconnecting from the network cuts off the editing workflow. It is not possible to access all the files in a project unless the host opens them.

Visual Studio Live Share

Visual Studio Live Share[3] is a plugin for Visual Studio Code and Visual Studio that enables sharing all files of a project loaded in the editor with someone else. In addition to that, it enables sharing debugging sessions and the ports opened by the debugging sessions are forwarded to clients. As with Teletype for Atom, files are only persisted on the "host" (See Figure 2.1.1).

Multihack-Brackets

Multihack-Brackets[4] is a plugin for the Brackets[5] editor. It enables sharing an entire folder structure. It requires a server. As of 13.3.2019, it is not possible to verify performance or functionality since joining a session crashes the brackets editor. The development seems to have stopped because the last commit in the repository was over a year ago (in February of 2018).

Codeshare

Codeshare[6] is a web-based collaborative editor. It is designed for interviews. The editor window offers syntax highlighting for a broad range of programming languages. One shared room always only contains a single file (SeeFigure 2.1.2).

---

[1]   https://github.com/atom/teletype/issues/211
[2]   https://teletype.atom.io/
[3]   https://visualstudio.microsoft.com/de/services/live-share/
[4]   https://github.com/multihack/multihack-brackets
[5]   http://brackets.io/
[6]   https://codeshare.io

2.1.2   Scientific Solutions

This section discusses scientific solutions for real-time collaboration.

CoVim

CoVim [5] uses operational transforms. Operational transforms, unlike CRDTs (See subsection 2.2.1), rely on shifting the positions of operations within the document to ensure consistency. Instead of observing user interactions, it observes changes to files and generates operational transforms from diffing states.

TouchDevelop

TouchDevelop[7] is an experimental web-based editor. It enables real-time collaboration by merging ASTs. The paper claims, that this approach can be generalized to a general-purpose language [20]. The AST translation needs to be implemented for every supported programming language. Additionally, the transformation from AST to text does not guarantee the same code which could lead to confusion for developers (See Figure 2.1.3).

CodeR

CodeR [12] is a web integrated development environment (IDE) with built in chat for the programming languages C, C++ and Java (See Figure 2.1.4).
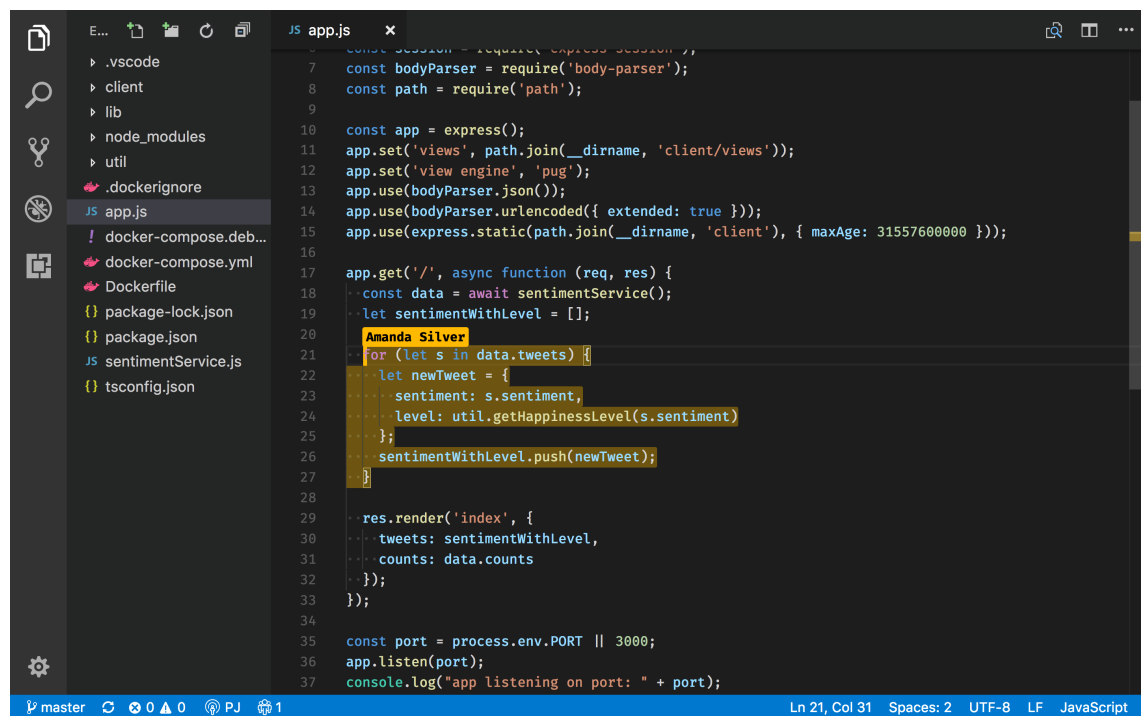
Figure 2.1.1: Visual Studio Live Share
https://code.visualstudio.com/blogs/2017/11/15/live-share

---

[7]   https://www.touchdevelop.com

Figure 2.1.2: Codeshare
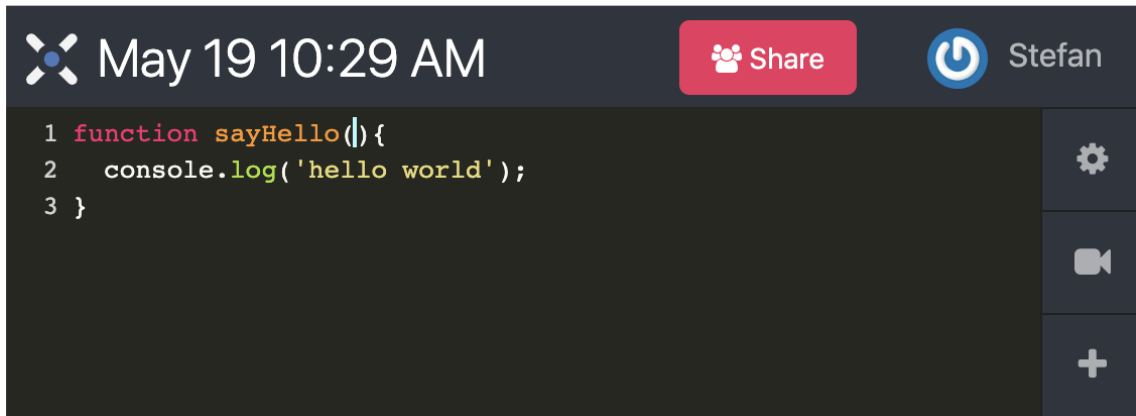https://codeshare.io

Collabode

Collabode is "a web-based Java integrated development environment" [8]. It shares changes between developers as soon as they have no compilation errors. It is designed for the Java language (See Figure 2.1.5).
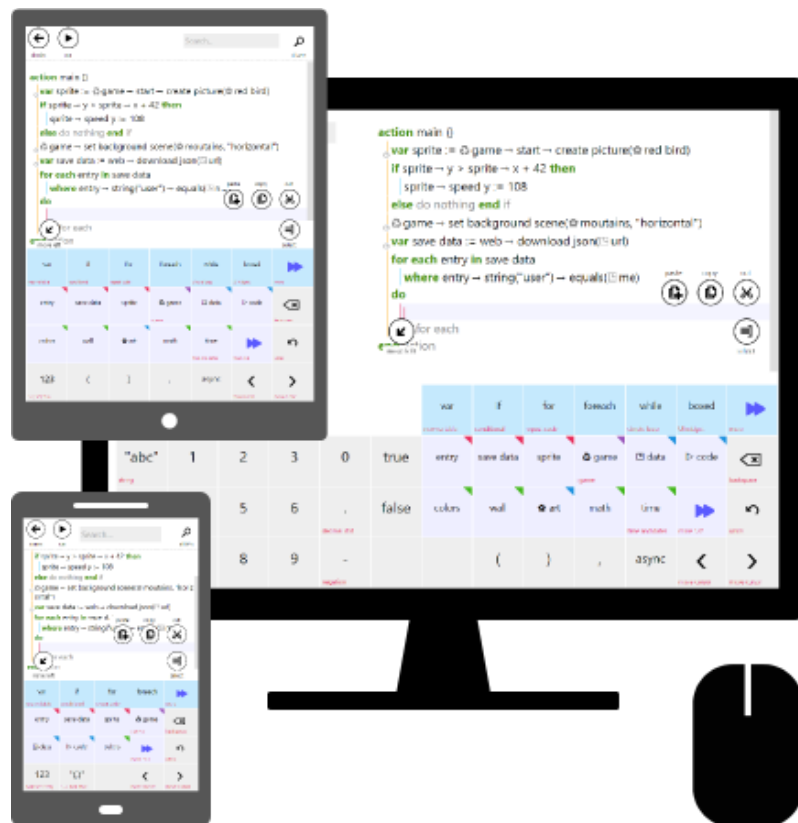
Figure 2.1.3: Microsoft TouchDevelop
https://www.touchdevelop.com/

Tool	Type	Location	Shared Content
Teletype for Atom	Extension	Host	Individual Files
CoVim	Extension	Host	Individual Files
Visual Studio Live Share	Extension	Host	Project Folder
Multihack Brackets	Extension	Distributed[a]	Project Folder
Codeshare	Web application	Server	Single File
Collabode	Web application	Server	Project Folder
CodeR	Web application	Server	Entire Project
TouchDevelop	Web application	Server	Entire Project

[a] Not possible to verify this information due to
     the editor crashing upon joining a session

Table 2.1.1: Overview state of the art

2.1.3 Summary

As described in Table 2.1.1, current solutions are either implemented as extensions to code editors or as web applications. Overall none of the current solutions have any considerations for dealing with an underlying version control system of a project.

2.2 Definitions

2.2.1 CRDT

To update shared objects stored at different sites a "commutative replicated data type" or CRDT is proposed by [19]. The idea is to design the underlying representation or data structure of edits to a document such that operations are commutative and therefore automatically converge at every copy of the document. However, there is a problem with this approach. If there are concurrent insertions at the same position of a document, a global order for the conflicting information has to

Figure 2.1.4: CodeR
[12]

Figure 2.1.5: Collabode
[8]

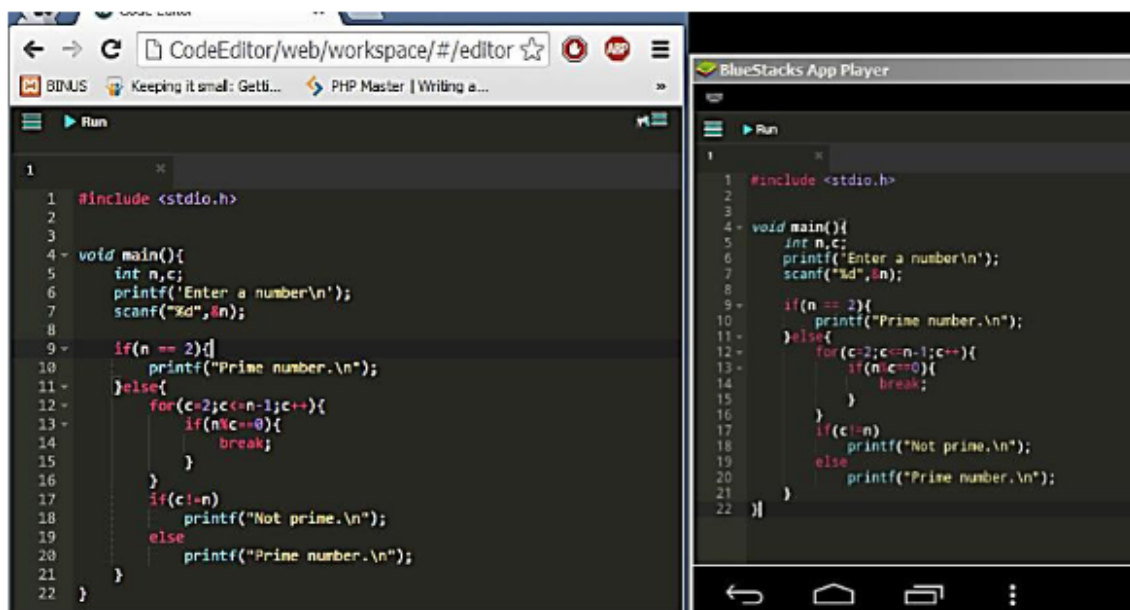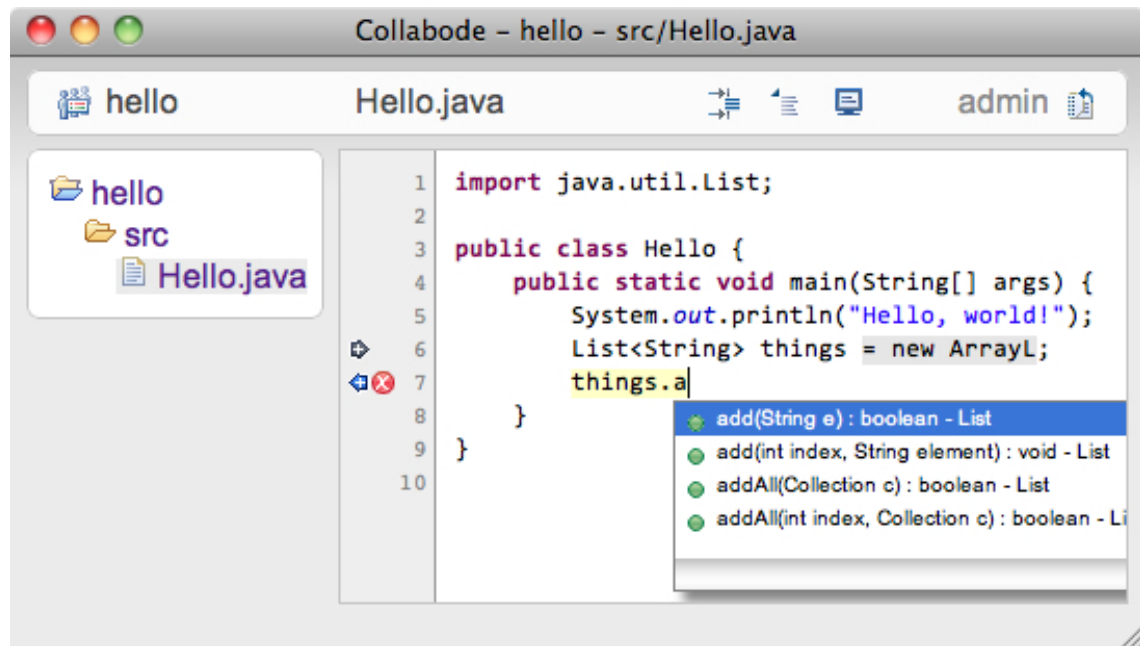be established. This problem has a solution: every site gets a unique identifier and a logical clock or counter. Concurrent inserts are then ordered either by the counter (smaller counter first) or if the counters are identical by identifier [19], [14].

2.2.2 Scrum

Scrum is an agile, iterative software development process. Work is divided into Sprints. There are daily meetings called "Daily Scrum". In those meetings, important questions are discussed in a group. One of them is "What impediments stand in the way of you meeting your commitments to this Sprint and this project?" [24], [25]. In theory, this introduces a maximum delay of one day from potentially blocking code being written to the person depending on it knowing about it. So if, for example a module is finished at 10am on one day the person depending on that module will learn about that at the next Daily Scrum meeting, about 20 hours later. This time can further be decreased by establishing tighter communication between team members. Especially in distributed development, it is important to "arrange an access to multiple communication tools" [17] for unofficial distributed meetings [16]. Since the version control system is a communication tool, improving version control systems improves communication tools and aids Scrum-based software development processes.

2.2.3 Code Review

Code review is used by most software development companies such as Google And Microsoft. The basic idea is that before merging a code modification onto the master branch, it is reviewed by someone else. This keeps code quality stable and educates developers [23], [3], [28]. This measure thesis aims to support these workflows while enhancing the developer experience.

The developers of CodeFlow reason that performance is key and that storing data on the device is an advantage to achieving better performance: "Another thing we focused on was performance. For that reason, even today CodeFlow remains a tool that works client-side, meaning you can download your change first and then interact with it, which makes switching between files and

different regions very, very fast." [6] Therefore, another important goal for this thesis is to deliver an experience that works client-side and keeps working when disconnecting and reconnecting to networks.

3 Methodological Approach

First, a state-of-the-art analysis was conducted. This analysis was looking into currently available open-source and commercial products for real-time collaboration. Additionally, the current literature related to real-time collaboration was studied. The papers mentioned in open source projects were used as a starting point for forward and backward literature analysis.

The following list of publishers were used as sources of literature: ACM Digital Library[1], IEEE Xplore Digital Library[2], SpringerLink[3], and ScienceDirect[4]. Google Scholar[5] was used as an additional search engine. Papers that are not published by the previously mentioned publishers were only used in exceptional cases and after consultation with the supervising assistant.

The literature survey was continuously documented in the following format:

```
1   <date >: <search query >
2     <publisher >
3       <paper title >
4   <search query > := one of the following :
```

- search terms

- conference

- forward/backward search paper title

- authors of specific papers

If multiple searches occurred on the same date, the date has to be included again for every search query.

Based on problems mentioned in papers and issues of projects from the state-of-the-art analysis, requirements for the extension were defined. These requirements are user-centred. Functional as well as non-functional aspects such as performance were defined as requirements.

Use cases for the extension were also defined. These use cases directly related to editing source code have a defined number of users as well as file sizes to be verifiable. Every use case includes at least the following information:

- Actor (the stakeholder)

- System (the software the actor is interacting with)

- Action (the goal of the actor)

---

[1] https://dl.acm.org/
[2] https://ieeexplore.ieee.org/
[3] https://link.springer.com/
[4] https://www.sciencedirect.com/
[5] https://scholar.google.com/

A VS Code[6] extension better suited to accommodate the identified use cases were implemented. The extension meets the following requirements:

- The extension targets VS Code 1.32 or newer and is written in Node.js

- Automated testing is used wherever possible and meaningful.

- The source code is commented and well documented.

The thesis was written in parallel with the development of the extension. Git was used as a version control system for the thesis as well as the extension.

The solution was evaluated by comparing the extension and the state-of-the-art tools in terms of suitability for the specified use cases. The evaluation is based on the amount of time required to fulfill a specific use case. Furthermore, any limitations compared to state-of-the-art solutions have been documented.

---

[6] https://code.visualstudio.com/

4 Requirements Analysis

Based on the limitations of current solutions, identified in chapter 2, this chapter defines requirements for the extension.

4.1 Requirements

This section describes the identified requirements for the extension.

4.1.1 Share the entire project structure

Every user should be able to see all changes to the project. Based on an issue and the discussion on the Teletype Github page explaining, that it would be a substantial improvement if the entire project structure was shared[1]. Being able to edit the entire project structure is a very important feature.

4.1.2 Only display changes in the same Git branch

Given that a lot of developers are using prototype branches and a significant number are using feature branches Git branches are a good indication that a specific problem is being worked on [18]. A prototype branch is a branch that is used to extend or improve software in an experimental workflow. Multiple implementations of features can be prototyped and later finalized in seperate branches without disturbing other people working on the project. Therefore, only displaying concurrent edits on the same edit removes noise of unrelated edits if features are developed in seperate branches.

4.1.3 Stage changes by author

Synchronizing changes to all developers introduces a problem: "[...] This means that git only becomes a way to have a backup as all the work is done using P2P! [...]"[2] Possibly unrelated modifications would be bundled into huge commits. In order to mitigate this, users should be able to stage changes by the author.

4.1.4 Retrieve required information from Git

As Git already contains information about the project and the author, the user should not have to enter this information into the extension again. Instead, the extension should, whenever possible, read this information (such as the current username) from Git.

4.1.5 Respect .gitignore

Files explicitly excluded from the version control system via the .gitignore file[3] should not be synchronized with other clients as these files might contain automatically generated files that de-

---

[1] https://github.com/atom/teletype/issues/211
[2] https://github.com/atom/teletype/issues/211#issuecomment-478306010
[3] https://git-scm.com/docs/gitignore

pend on the local system configuration or contain sensitive information. This solution was also proposed in the Teletype for Atom GitHub Issues[4]

4.1.6 Performance

Although VS Code runs extensions in seperate threads to prevent performance impacts of misbehaving extensions.[5]

Performance of the extension should be good enough that typing on two computers with one hand each should be possible without introducing errors or noticeable delay if both computers are connected to the same network via ethernet.

To better understand the requirement stated in the previous paragraph consider this scenario: A person is sitting in front of two computers both with VS Code and the extension open. The left hand is on the keyboard of the first computer, the right hand on the keyboard of the second computer. The person should be able to type a sentence without errors being introduced by delays in change synchronization.

4.1.7 Disable foreign changes

In order to compile source code, it is mandatory that the code is free of syntax errors and does not change during compilation. Therefore, it should be possible to disable receiving changes from other clients. Sometimes it might even be necessary to roll back the changes other people have made to the codebase.

4.1.8 Support bad internet connections

The extension should be able to support the client losing the internet connection. Even if the user relaunches the code editor while offline, the changes of other users should still be as they were when the connection was lost. As soon as connectivity is restored, new changes should start to be displayed.

4.2 User stories

This section describes user centered requirements in the form of user stories.

1. As a programmer, I want to access the entire project structure when using a code editor.

2. As a programmer, I want a clean separation between branches and do not want to see changes to other branches when using a version control system.

3. As a programmer, I want to be able to stage changes that I made to Git.

4. As a project manager, I want to see who made a specific modification to a project in the version control system.

5. As a programmer, I do not want to configure a second version control system when I already provided this information to Git.

6. As a programmer, I do not want files covered by my .gitignore configuration to be shared with others.

---

[4] https://github.com/atom/teletype/issues/211#issuecomment-376999575
[5] https://code.visualstudio.com/api/advanced-topics/extension-host

7. As a programmer, I want to be able to edit files with up to 4 people at a time.

8. The system processes files up to 30.000 characters.

4.3 Scenarios

In this section, multiple scenarios are derived from the user stories.

4.3.1 Scenario 1

Person A and B are working on a Node.js Express project together. Person A is working on branch "Ba", and person B is working on Branch "Bb". Person A has edited the file "/app.js" (paths are described as absolute from the project root) as well as the file "/routes/account.js" on "Ba". Person B has edited the file "/app.js" as well as the file "/routes/main.js" on "Bb". Person A encounters unexpected behaviour of his code. He requests help from person B. Neither persons A nor B want to commit their changes at this point. Person B uses the VS Code Command Palette and chooses the option "Procurrently: Checkout Branch" and chooses branch "Ba". Person B now sees only the modifications in "/app.js" and /routes/accounts.js from branch "Ba". Upon discovering the problem in the source code, person B modifies "/routes/account.js" to solve the problem. Person A can see the changes in real-time. Person B once again opens the VS Code Command Palette and chooses the option "Procurrently: Checkout Branch" and chooses branch "Bb". Person B now sees the modifications in "/app.js" and "/routes/main.js" on Branch "Bb" and can continue working on Person B's task.

4.3.2 Scenario 2

Person A and B are working on an Node.js Express project together. Both of them are working on branch "master" in the "/app.js" file. Person A is modifying and testing a new feature while person B is working on setting new HTTP caching headers for the Express app. This is interfering with the test of person A. Person A opens the VS Code Command Palette and chooses the option "Procurrently: Toggle remote changes". All the modifications of B are reverted in the documents of person A. During that person B can still see the modifications of person A which helps person B to determine the correct caching headers for the different routes. Person A's laptop disconnects from the network. Therefore, person B can no longer receive real-time updates from person A. Person A's laptop reboots unexpectedly. Once the laptop is booted up again, person A opens VS Code and can continue to work where person A left off. Person A now wants to know about the progress of person B and turns on remote changes. Person A can see all the changes of person B until person A disconnected from the network. Person A disables remote changes again and continues to work on his new feature. Later on, person B's laptop reconnects to the network and person B can receive all the modifications person B missed. Once person A is happy with the implementation of the new feature, person A wants to test the caching behaviour himself to make sure person B has understood his code correctly. Person A opens the VS Code Command Palette and chooses the option "Procurrently: Toggle remote changes". Person A now sees all the modifications of person B again including the modifications B made while person A had disabled the remote changes.

5 Implementation

This chapter describes the implementation details of Procurrently, the extension for VS Code. It gives an overview of the Git library, that needed to be developed since other available solutions do not work inside a VS Code extension context. Furthermore, the used VS Code APIs are listed. Additionally, the data structure and change handling by the extension are described.

5.1 Git

Git provides useful information about the current project, the developer is working on, such as:

- The current branch (the problem being worked on)

- The root directory of the project (for path resolution across devices)

- The current version of a file (as a basis for collaborative edits)

- Information about the developer (the username)

Git stores this information in the ".git" directory in the root folder of the project. The data is stored in a compressed format. In order to access the information, a small library using low level Git commands was implemented. A custom implementation was necessary because the NodeGit[1] library crashed VS Code as soon as the extension imported it.

The Git library provides a low-level interface for interacting with a Git repository. It provides functions to find the Git directory for a given file (given the file is inside a Git repository) and retrieve information like the path of a file within the repository, information about the current commit, hash, branch, remote URL, username of the repository, as well as different versions of a file. This information is crucial for determining the version of a file that has been modified by the user and finding the identical file on the machine of another user.

Additionally, the Git library provides a function to stage files for commit, commit changes and do a Git reset. The stage and commit functions are required in order to stage changes by author and commit them. The Git reset function is used to synchronize file contents to the version known to Git. This enables synchronizing the files without establishing base versions of files over the network first.

The Git library can invoke a callback function when the current state of a repository changes. This is the case if a different branch is checked out or the current commit changes due to the user committing or a git pull. Because once the repository changes, the appropriate changes have to be applied to documents and only changes for the same repository state can be replayed.

---

[1] https://github.com/nodegit/nodegit

5.2   teletype-crdt

"The string-wise sequence CRDT powering peer-to-peer collaborative editing in Teletype for Atom."[2] This library will be used for tracking changes. It is written in JavaScript and currently does not include an API documentation.

Teletype-crdt provides the Document class. This class represents a shared document using a CRDT. In order to notify the document representation about a local change, the setTextInRange (see Listing 5.1) function can be used.

When a change from another instance of the document is received, the teletype-crdt document can be notified using the integrateOperations (see Listing 5.2) function. This function returns a set of TextUpdates, which are changes to the text of the document to match the operations.

Since the document turns every text change into an operation and operations contain information about the author, sometimes the effect of operations needs to be determined. For example, if staging changes by author. To do this, the undoOrRedoOperations function is used (see Listing 5.3). It returns a set of modifications for the file on disk equivalent to an undo of a set of operations. Which, if this function is not used otherwise translates to the mapping of operations to effects of operations described above.

```
1  setTextInRange(start: {row:Number, column:Number}, end: {row:Number, column:
   ↪ Number}, text: string, options?: any): [operation];
```
Listing 5.1: teletype-crdt setTextInRange

```
1  integrateOperations(operations: [operations]): {textUpdates:[textUpdate],
   ↪ markerUpdates:any};
```
Listing 5.2: teletype-crdt integrateOperations

```
1  undoOrRedoOperations(operationsToUndo: [operation]): any;
```
Listing 5.3: teletype-crdt undoOrRedoOperations

5.2.1   Git and teletype-crdt

As a basis for the edit history the current Git commit in the current branch will be used. Teletype-crdt uses numeric siteIds to identify changes by author. The current Git commit is imported as edited by siteId 1 upon discovering the file. By doing this, all clients have an initial shared state based on the current version of the file known to Git.

5.3   VS Code Extension API

VS Code runs extension in a separate process and provides an asynchronous JavaScript API. The examples provided in the vscode-extension-samples repository[3] are mostly written in Typescript[4] and all the Interfaces have type definitions for Typescript. Therefore, the extension will use Typescript as well.

---

2   https://github.com/atom/teletype-crdt
3   https://github.com/Microsoft/vscode-extension-samples
4   https://www.typescriptlang.org/

5.3.1 Used API Functions

The VS Code API provides a set of asynchronous functions that return Promises. Promises have been added in ECMAScript 6 and provide a standardised way of handling asynchronous code in ECMAScript. [13] ECMAScript is the standard for the JavaScript language.

Procurrently mainly uses the onDidChangeTextDocument (see Listing 5.4) and the applyEdit (see Listing 5.6) function.

Using the onDidChangeTextDocument function (see Listing 5.4), a callback function is invoked whenever a text document is changed.[5] This callback function converts the event provided by VS Code to a teletype-crdt operation using the setTextInRange function (see section 5.2) and sends the resulting operation to all other peers.

Other peers subsequently use the applyEdit function (see Listing 5.6) to apply the TextUpdates from the integrateOperations function (see section 5.2) to the document on disk.

```
1   vscode.workspace.onDidChangeTextDocument()
```
Listing 5.4: VS Code API onDidChangeTextDocument

```
1   vscode.workspace.onDidOpenTextDocument()
```
Listing 5.5: VS Code API onDidOpenTextDocument

```
1   vscode.workspace.applyEdit(edit)
```
Listing 5.6: VS Code API applyEdit

The applyEdit function (SeeListing 5.6) returns a promise which resolves when the change has been added to the text document. An edit is given by a document, the line and column of start and end of the edit in that document and the new text to be inserted there. If text is added or removed within a line, the columns of the text change after the edit has been applied (when the promise resolves). If another edit is applied before the promise resolved, the indices of the text might have not been changed yet and therefore at a different position than expected. VS Codes WorkspaceEdit supports grouping change operations together. But change operations are not treated as happening all at a time so if for example the text "123" was typed the teletype-crdt library would essentially output Line 1 Char 1 to Line 1 Char 1 changed to "1", Line 1 Char 2 to Line 2 Char 2 changed to "2" and so on. This is kind of expected behaviour so far but VS Code's WorkspaceEdit treats this as independent operations and if there was text after the insertion the changes would not be inserted as one block but interlaced with the previous text.

In order to avoid this race condition all changes to a file have to be carried out sequentially. This is not noticeable when changes by regular typing are integrated into the local document. But when the multiple cursors[6] are used the change integration can slow down noticeably. In order to support multiple cursors, the changes are sorted by column in descending order to prevent shifting indices. (See Listing 5.7)

```
1   textUpdates.sort((a, b) => b.oldStart.column - a.oldStart.column)
```
Listing 5.7: Sorting Changes by Column to Prevent Index Shifting

---

[5] https://code.visualstudio.com/api/references/vscode-api#workspace
[6] https://code.visualstudio.com/docs/getstarted/tips-and-tricks#_editing-hacks

---

5.3.2 Tree View

In order to stage changes by author, a Tree View listing the Authors, who made changes, was created. The container for a Tree View needs to be defined in the package.json file. Figure 5.3.1

```
1  "contributes": {
2      "viewsContainers": {
3          "activitybar": [
4              {
5                  "id": "change-explorer",
6                  "title": "Change Explorer",
7                  "icon": "media/icon.svg"
8              }
9          ]
10     }
11     [...]
12  }
```

Listing 5.8: Tree View Activitybar

First an entry in the activity bar has to be declared in the viewsContainers section (See Listing 5.8). It defines the icon as well as the hover text (called title) of the tab in the activity bar.

```
1  "contributes": {
2      [...]
3      "views": {
4          "change-explorer": [
5              {
6                  "id": "contributors",
7                  "name": "Contributors"
8              }
9          ]
10     }
11  }
```
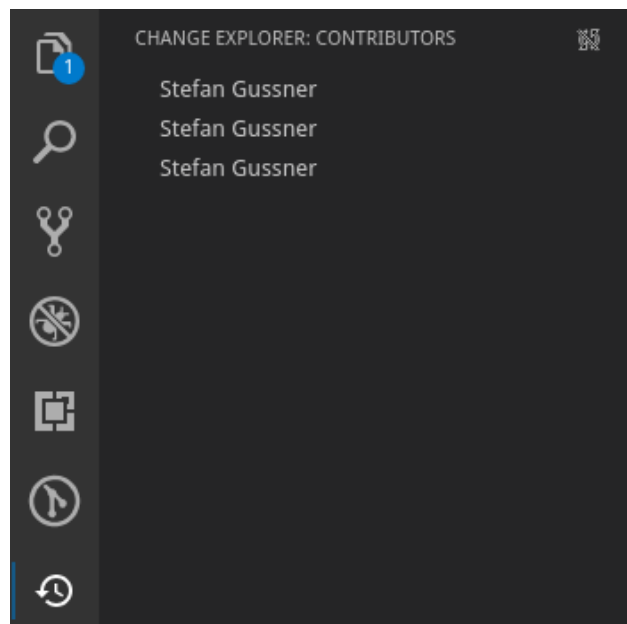
Listing 5.9: Tree View Panel Definition

Figure 5.3.1: Tree view

Additionally, the view has to be declared.Listing 5.9 This defines the heading for the Tree View.

```
1 const treeview = new ContributorsTreeView(crdt.getUsers);
2 vscode.window.registerTreeDataProvider('contributors', treeview);
```
Listing 5.10: Define Tree View Data Provider

To populate the Tree View with data, a TreeDataProvider has to be registered to the view id. (See Listing 5.10) The TreeDataProvider interface defines functions that return all items for the Tree View as well as refresh the content of the Tree View.

5.4 Data Model

```
1 const documents = new Map<string, { document: Document, metaData: { commit:
  ↪ string, branch: string, repo: string, file: string, users: Map<Number,
  ↪ string> } }>();
2 /** remote repository to path mapping */
3 const localPaths = new Map<string, string>();
4 /** The current branch for a file */
5 const branches = new Map<string, string>();
```
Listing 5.11: Data Model Declarations

All the data is stored in a map called documents. (See Listing 5.11) It's key is composed of the filename and a specifier composed from the commit, branch and remote/origin repository URL.

In order to keep track of the current branch and head commit of files, the branches map contains the identifier for a file by file path.

The localPaths map contains mappings from remote/origin URLs to local Git directory locations. This enables keeping track of files across different projects.

The documents map values contain

5.4.1 The Document Object

The document object has two properties

- document

- metaData

The document property points to an instance of the document class provided by teletype-crdt.

The metaData property contains the relevant information from Git about the document:

- branch

- commit

- repo

- file

- users

The users property contains a map with the teletype-crdt siteId as a key and the Git username as a value. This information is required to display usernames in the staging Tree View.

5.5   Reacting to Local Changes

If a file has not yet been accessed, it has to be registered. This establishes the current version of the file known to Git. Additionally The localPaths map is updated with the repository remote origin URL as the key and the location of the git repository on disk as the value. This will later be used for incoming changes.

In order to process a local change, provided by the onDidChangeTextDocument API call, the extension checks, if the change has been added by a remote client. This is necessary because the VS Code API does not differentiate between changes by the user and changes by extensions. (See Listing 5.12) Otherwise changes are duplicated endlessly because every remote change is propagated back to all other clients as a new change.

```
1  const objects = ['start', 'end'];
2  const props = ['line', 'character'];
3
4  //check if this change has just been added by remote
5  const knownChanges = currentChanges
6      .filter(c =>
7          objects.map(o =>
8              props.map(p =>
9                  c[o][p] == change.range[o][p]))
10          && c.text == change.text
11          && c.filepath == e.document.fileName);
12 if (knownChanges.length > 0) {
13     //remove from known changes
14     currentChanges.splice(currentChanges.indexOf(knownChanges[0]));
15 }
```

Listing 5.12: Is This Change Already Known to The Data Model?

To update the teletype-crdt document the setTextInRange function is used. It returns a list of operations. This list of operations is sent in a JSON object containing the metaData associated with the document. (See Listing 5.13)

```
1  {
2      "update": {
3          "metaData": {
4              "branch": "refs/heads/master",
5              "commit": "05fc4663235f36ba054ea37fd7f92e9a5555edf2",
6              "repo": "git@bitbucket.org:company/a_repository.git\n",
7              "file": "/app.js",
8              "users": {}
9          },
10         "operations": [
11             {
12                 "type": "splice",
13                 "spliceId": {
14                     "site": 2766400253437581,
15                     "seq": 3
16                 },
17                 "insertion": {
18                     "text": "a",
19                     "leftDependencyId": {
20                         "site": 0,
21                         "seq": 0
22                     },
23                     "offsetInLeftDependency": {
24                         "row": 0,
25                         "column": 0
26                     },
```

```
27                    "rightDependencyId": {
28                        "site": 2766400253437581,
29                        "seq": 1
30                    },
31                    "offsetInRightDependency": {
32                        "row": 0,
33                        "column": 0
34                    }
35                }
36            }
37        ],
38        "authors": [
39            [
40                2766400253437581,
41                "Stefan Gussner"
42            ]
43        ]
44    }
45 }
```

Listing 5.13: Network Data Packet

5.6 Network Transport

An essential part of Procurrently is the handling the changes over the peer to peer connections.

5.6.1 Establishing the Peer to Peer Connections

When the extension is activated, a to a bootstrap server is established. The peer sends the bootstrap server his IP address as well as the port of the socket, the peer is listening on for incoming connections. Upon receiving this information, the bootstrap server responds with a list of IP addresses and ports of all the other registered peers. The peer then tries to connect to every peer in that list. This is illustrated in Figure 5.6.1.

Once a connection is established, the peers exchange all the changes they know about.

5.6.2 Handling Changes

Every change is propagated to every node known to the peer. This approach is performant enough for small groups. To scale up to more clients, peers could intelligently only send changes to peers, who are currently on the same branch and peers could request operations from other peers when switching branches. Additionally, changes could be propagated via a gossip-based protocol.

5.7 Handling Remote Changes

Handling incoming changes has two main challenges:

- concurrent changes

- finding files on disk

5.7.1 Concurrent Changes

Concurrent changes can introduce errors when adding them to a file. Given two changes on the same line one of the changes will have its index changed by the other one.
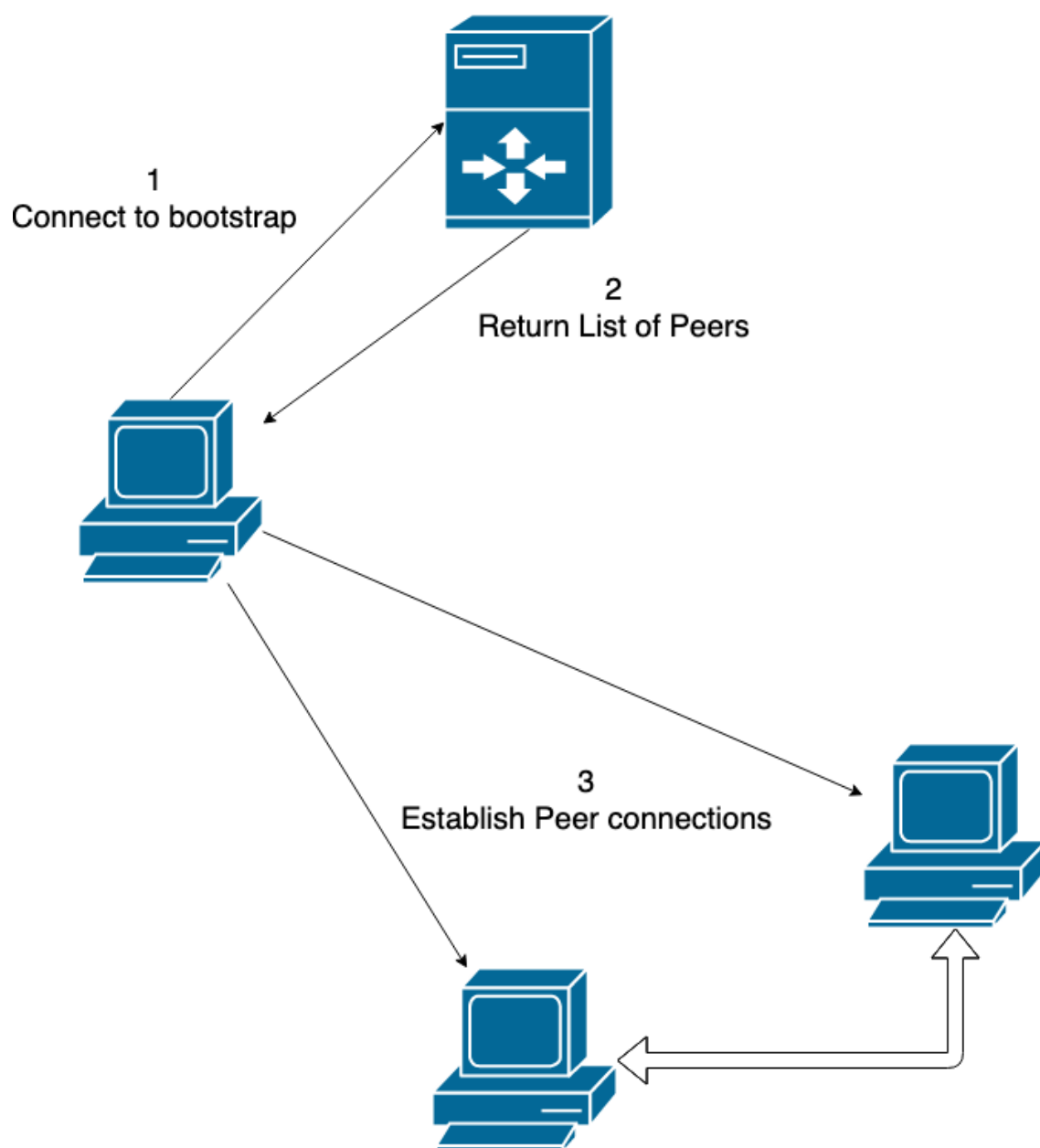
1
Connect to bootstrap

2
Return List of Peers

3
Establish Peer connections

Figure 5.6.1: Establishing Peer to Peer Connections

To illustrate the problem, consider this example:

The initial line consists of the string "12345". Now peer A adds "g" after index 3 and peer B adds "c" after index 4 The local result for A is "123g45" and the local result for B is "1234c5". The correct result for the changes would then be "123g4c5" The crdt data type handles this concurrency problem and if the change from peer B is processed after the change from peer A the insert operation of B is adjusted from index 4 to 5. But this assumes that all the changes are processed sequentially. If the order of insertions is not guaranteed the resulting string could turn into "123g45c" if the crdt document processes the changes in the order A -> B and the VS Code Edit is processed in the order B -> A.

To ensure consistency, the network layer waits for the change handling promise to resolve before processing the next change. This can be thought of as "pretending network changes have been delayed". CRDTs such as atom-teletype are designed to handle delayed network packets. Therefore, this ensures consistent change replication. Listing 5.14 ensures that incoming change packets are processed sequentially by building a promise chain.

```
1  this._currentEdit = this._currentEdit.then(() => {
2      [...]
3      return this._onremoteEdit(receieved.update);
4      [...]
5  })
```

Listing 5.14: Network Promise Chain

5.7.2 Locating Files on Disk

Given the information from the network locating the file the change corresponds to is not trivial.

First the appropriate Git repository has to be located. This information can be looked up in the localPaths map.

If the file has not been accessed previously, the base version of the file has to be established. By using the getCurrentFileVersion function of the Git library to retrieve the appropriate version from the Git repository to replay changes onto.

5.7.3 Adding Changes to File

If the file changed by the remote peer is currently checked out, the change has to be incorporated into it. Otherwise the change will just be saved to the atom-teletype document representation. (See Listing 5.15)

```
1  //only apply changes if on the same branch and remote changes visible
2  if (branches.get(filepath) == getSpecifier(metaData.commit, metaData.branch,
   ↪  metaData.repo) && remoteChangesVisible) {
3      const doc = getLocalDocument(filepath).document;
4      const textOperations = doc.integrateOperations(operations);
5      try {
6          await pendingRemoteChanges.then(() => applyEditToLocalDoc(filepath,
           ↪ textOperations));
7      } catch (e) {
8          console.error(e);
9      }
10 } else {
11     //save changes to other files
12     getLocalDocument(filepath, metaData.commit, metaData.branch, metaData.
       ↪ repo).document.integrateOperations(operations);
13 }
```

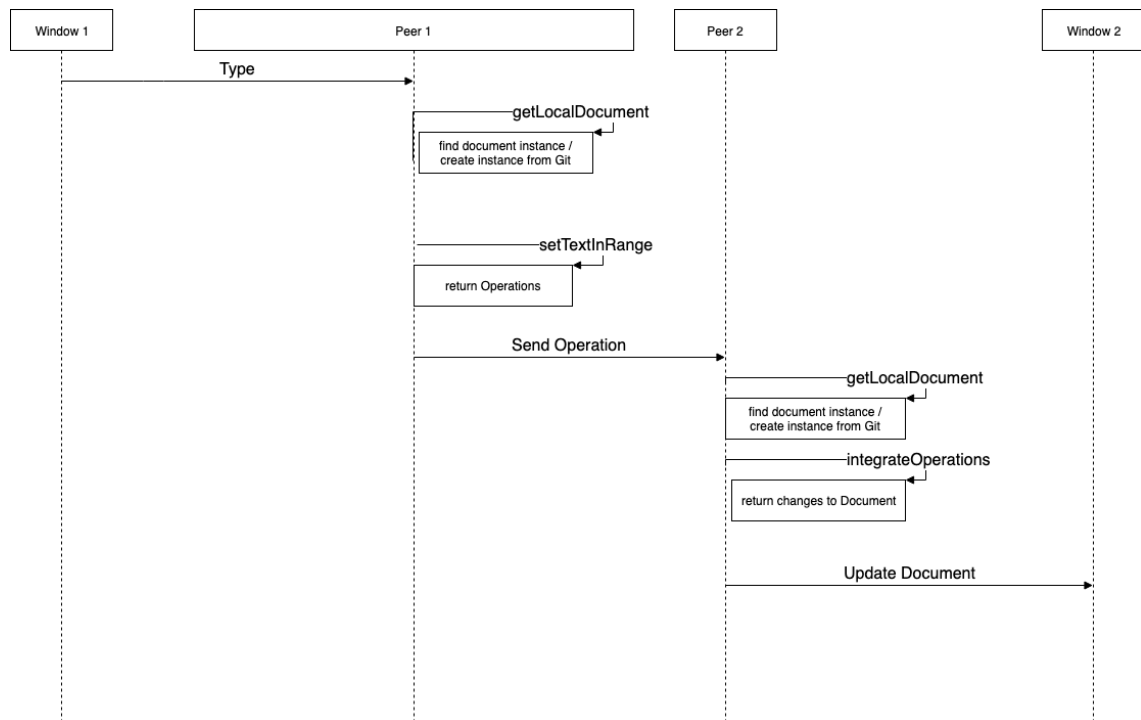**Listing 5.15:** Adding Change to Local Document

Figure 5.7.1: Overview - Handling Edits

Figure 5.7.1 gives an overview of how changes are processed.

6   Evaluation

In this chapter Procurrently will be evaluated based on the scenarios and compared to existing tools. Furthermore, limitations compared to other tools will be described.

6.1   Evaluating Scenarios

Procurrently was evaluated using the scenarios described in section 4.3. These were tested on a real git repository containing 600 files. One of the files most commonly used in testing was app.js containing over 7000 characters or about 200 lines. The test was run on two MacBook Pros using an 802.11ac Wi-Fi connection. The bootstrap server was run on one of the MacBooks. The IP for the bootstrap server was configured before starting the test using the VS Code settings menu.

6.1.1   Scenario 1

The branches used in the test were "bug-hunt" and "buildsystem-test". On branch bug-hunt the files /app.js (see Figure 6.1.1) and /routes/account.js (see Figure 6.1.2) were modified by inserting comments. The configured username was "Stefan Gussner". After that on branch "buildsystem-test" the files /app.js and /routes/index.js were modified by inserting comments. The configured username was "Michael".
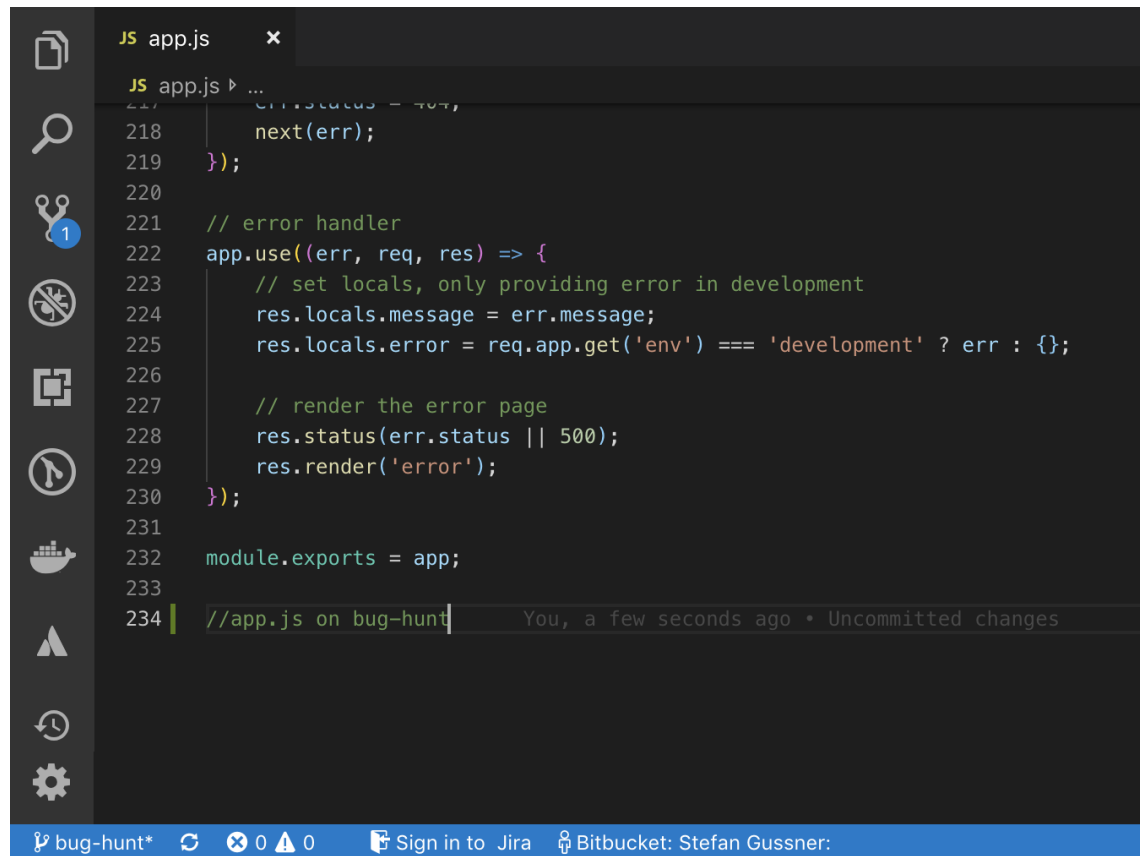
On the computer with the Username "Michael" configured, the VS Code Command Palette was opened and the option Procurrently: Checkout Branch was selected (see Figure 6.1.3). From the presented options the branch "bug-hunt" was selected (see Figure 6.1.4). All the files were closed and the files /app.js and /routes/account.js were opened. The modifications on branch "bug-hunt" were visible in the files and the changes in branch "buildsystem-test" were no longer visible. The file /routes/account.js was modified by the computer with the Username "Michael" inserting another comment (see Figure 6.1.5). The changes were immediately visible on the other computer. Using the VS Code Command Palette, the computer with the Username "Michael" configured switched back to branch "buildsystem-test".

6.1.2   Scenario 2

Both computers started out on the same branch. On the computer with the username "Stefan Gussner" a comment reading "comment from stefan" was created. On the other computer a comment reading "comment from michael" was added (see Figure 6.1.6). The VS Code Command Palette was used to execute the Procurrently: Toggle remote changes option (see Figure 6.1.7). After that, all remote changes were removed from the file and only local changes were left (See Figure 6.1.8). Later the Command Palette was used again to execute the Procurrently: Toggle remote changes command again to display the remote changes again.

6.2   Results

Procurrently succeeded in using information provided by Git to provide branch based real-time document synchronization. Only changes on the same branch are displayed to other users.

Figure 6.1.1: Scenario 1: Branch bug-hunt initial modifications app.js

Changes to all documents in Git projects are shared with other peers (note that this is not the entire project structure as described in section 6.3).

Changes can be staged by author. This provides accountability and clarity about changes. Figure 6.2.1 shows staging a change by author using the Tree View panel.

Files defined to be ignored by Git are also ignored by Procurrently. This prevents accidentally publishing classified information and conflicts introduced by generated files.

6.2.1   Other Solutions

Procurrently runs on the client and to some extent even works offline. Although some of the Tools described in section 2.1 are client-side applications/extensions none of them provide support working offline. Having data available on the client at all times provides a performance advantage for Procurrently because when changesets have to be computed for displaying/hiding remote changes or branches are switched all the necessary data is already in memory on the device and is therefore very fast. It also enables real-time collaborations in environments without a stable internet connection such as trains.

Other solutions do not consider the version control system when providing real-time changes. This limits their usability in corporate environments. Procurrently is a tool for Git based environments and their use-cases in corporate environments.
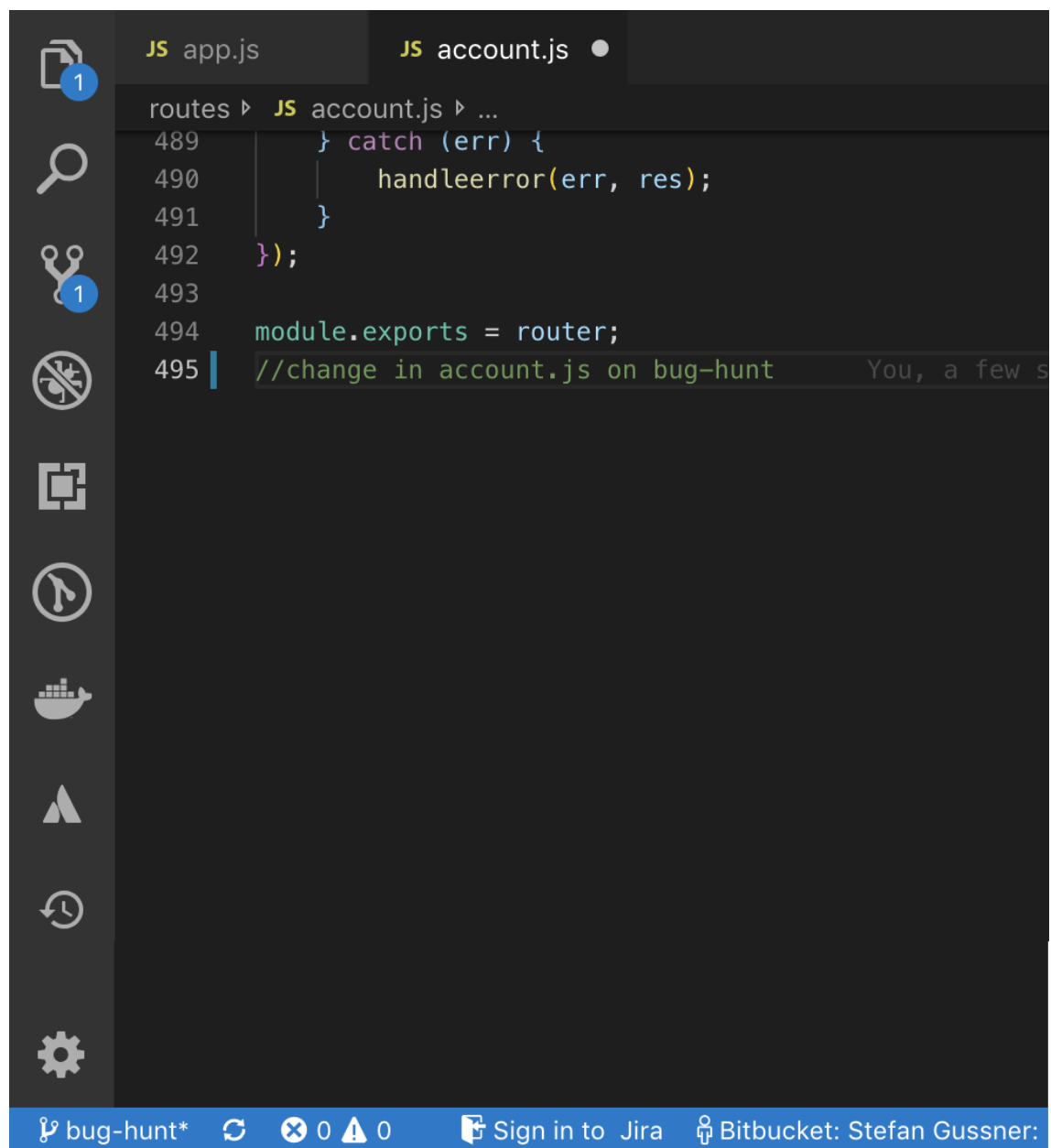
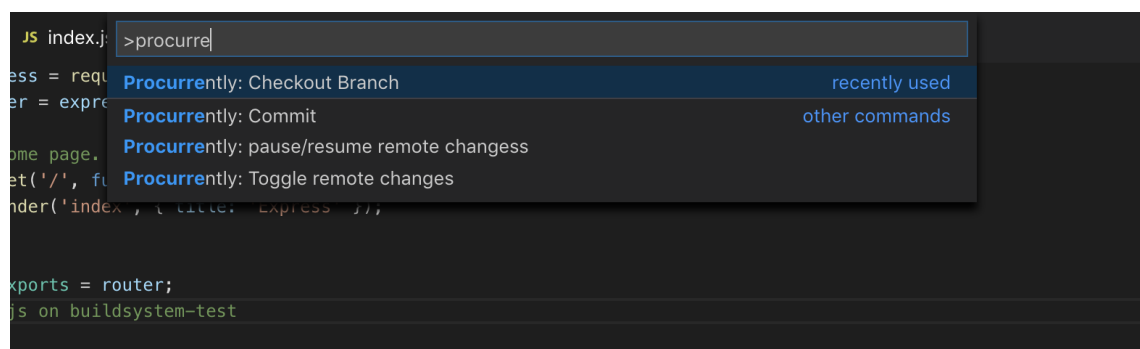**Figure 6.1.2:** Scenario 1: Branch bug-hunt initial modifications routes/account.js
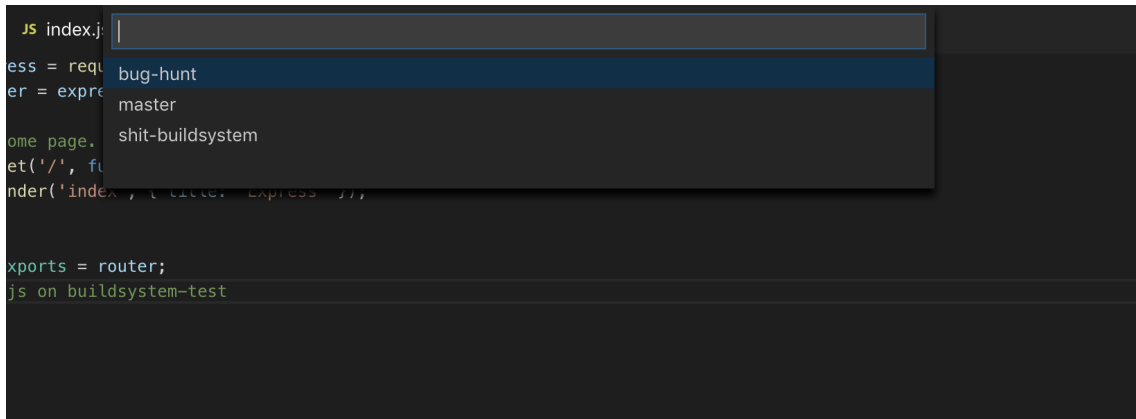

Figure 6.1.3: VS Code Command Palette Menu
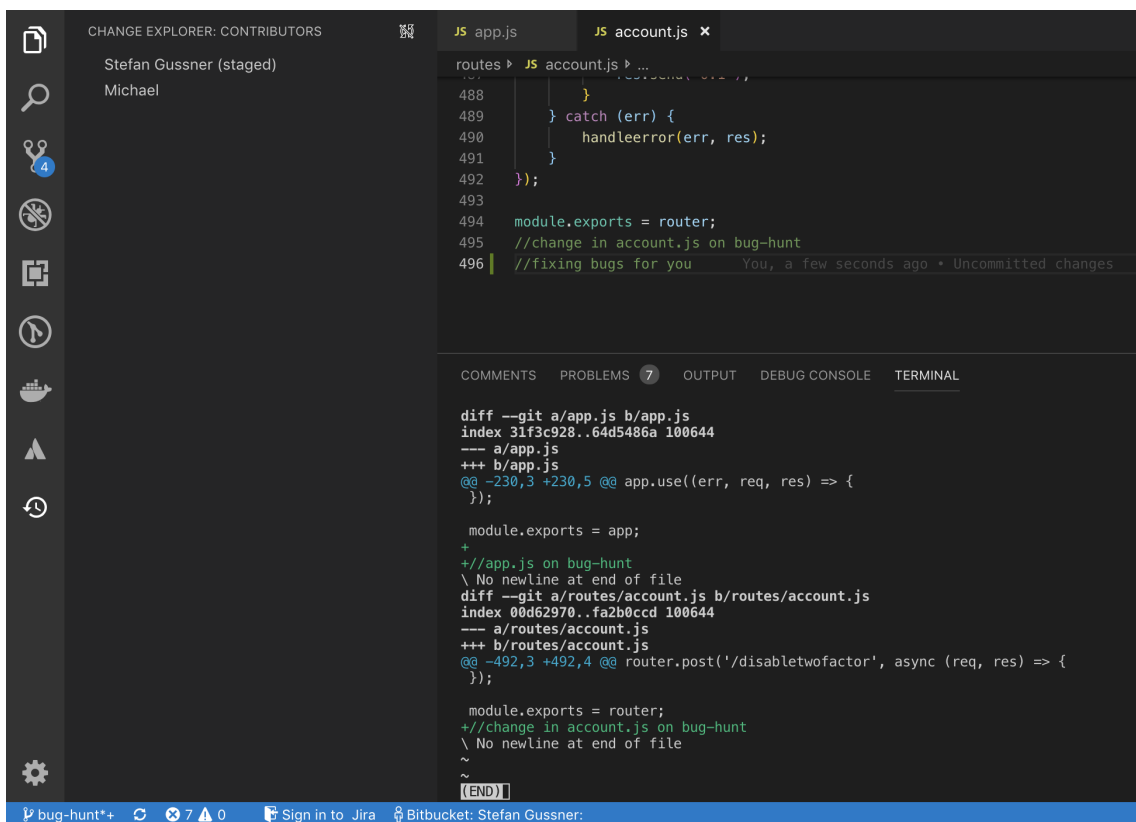
Figure 6.1.4: Select Branch

Figure 6.1.5: Scenario 1: account.js modified by Michael

6.2.2   Testing Procurrently in the Real-World

Procurrently was tested in a real-world development environment with 2 users for 5 hours. After a short introduction to the new Git checkout procedure the tool did not cause friction for the users. During the test situations similar to subsection 4.3.1 were encountered multiple times. Procurrently was working as expected and aided the problem solution by providing fast context switching without the overhead of synchronizing changes via Git.

Figure 6.1.6: Scenario 2 - Comments Added by Both Collaborators

Figure 6.1.7: Scenario 2 - VS Code Command Palette Menu Toggle Changes

6.3 Limitations

Compared to other current solutions, Procurrently is unable to track files that do not exist in the last Git commit because the base version of a file is derived from the latest Git commit. This behaviour is necessary because some VS Code extensions create temporary files that do not get persisted to the filesystem but fire change events. Not ignoring these files would force developers to extend their .gitignore configuration and interfere with Procurrentlys goal of not requiring extra configuration by the developer.

Figure 6.1.8: Scenario2: Only Local Changes Visible

Procurrently is not secure. Anyone on the network can modify files in the repository. Also, all the traffic is unencrypted and can therefore be read by anyone on the network. This might not be a problem for open source projects as well as people working in corporate network environments.

The staging view does not differentiate between different branches so all collaborators across all Git repositories and branches are presented as changes to be staged instead of filtering by collaborators who changed the current working context.

Due to the possible inconsistencies introduced by the asynchronous VS Code applyEdit API function and the workaround described in subsection 5.7.1 processing huge change sets is noticeably slow. This is most obvious when processing multi cursor changes with more than 20 cursors at once.

Because the networking layer does not support UDP hole punching [10] or similar techniques, communication between peers only works when clients are in the same local network.

Branch switching and committing changes sometimes have to be done through custom commands in VS Code instead of having the flexibility of doing those tasks using the command line and potentially using other programs or scripts to call those functions.

Procurrently enables switching branches while the current branch has been modified and the changes would be overwritten by Git. But this feature only works when using the custom VS Code command "Procurrently: Checkout Branch".

Procurrentlys function to commit changes by author only works properly if using the custom VS Code command "Procurrently: Commit". Otherwise non committed changes will be lost after the commit because the changes are tied to the commit hash.

If there is no overlap between people working on a project, information about concurrent edits cannot be exchanged between peers. One possible solution might be to host a Procurrently nodes in the Cloud. As described in [26], Cloud hosted nodes could be for Procurrently what GitHub is to Git. Peer to peer technology has been used to improve the Cloud before. [21], [22], [2]. Cloud

Figure 6.2.1: Staging Changes by Author

technology can be used to improve peer to peer software. This concept is used in the experimental Beaker Browser[1]

---

[1] https://beakerbrowser.com/docs/how-beaker-works/peer-to-peer-websites#keeping-a-peer-to-peer-website-online

7 Conclusions

Procurrently demonstrates an approach to real-time collaboration compatible with traditional work-flows based on Git with opportunistic peer to peer communication.

Developers do not have to think about the extension. It is possible to disable or pause remote changes.

Developers can easily jump between branches without the need to commit work before doing so. This improves upon VCS use-cases such as helping a colleague with a problem without having to commit, push and pull changes first.

In contrast to other real-time collaboration solutions, accountability for changes is not lost when committing to a Git repository by staging changes by author.

7.1 Future Research

The network layer of Procurrently is implemented in a very simplistic way. Possible next research steps for more efficient network usage could be:

- Providing real-time updates only to peers on the same branch and batching and bundling updates for other peers

- Use 5G D2D communication for the network transport for peer to peer communication [27]

- Encrypt traffic and authenticate collaborators

Establishing an initial shared state might be possible with local changes present instead of doing a Git reset. This might require user interaction to resolve conflicts using 1-way merges or manually changing the file contents. [30]

Furthermore it is possible that a system like Procurrently could reduce friction in establishing group awareness by enabling group coding sessions over long distances instead of just relying on mailing lists and chats as described in [9] and would make it easier to know, "whom to contact" [11], [9].

As speed in Continuous Integration systems is crucial: "The dominant technical factors which explain Travis abandonment are **Build duration** and Language" [29], Procurrently could also find use-cases in Continuous Integration augmenting common workflows based on Git. Current workflows react to commits as described:

"

- Developers builds and pushes code to Github

- Github will use a webhook to notify Jenkins of the recent change

- Jenkins will pull the Github repository, including the Dock-erfile describing the image along with the application

- Jenkins will then build the Docker image of that application on the Jenkins slave node

- Jenkins will run the Docker container on the slave node and will execute the appropriate tests

- If the tests are successful, the docker image is then pushed to the Docker Hub with the appropriate version number

" ([7])

Additionally, to listing to pushes on GitHub, builds could be triggered when any developer changes a file in Procurrently, and the results of the test results could be displayed within the editor as soon as the tests have run. This could reduce feedback delay of Continuous Integration systems.

Bibliography

References

[1]   Brian de Alwis and Jonathan Sillito. „Why are software projects moving from centralized to decentralized version control systems?" In: *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering* (2009), p. 38.

[2]   Ozalp Babaoglu, Moreno Marzolla, and Michele Tamburini. „Design and Implementation of a P2P Cloud System". In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC '12. Trento, Italy: ACM, 2012, pp. 412–417. ISBN: 978-1-4503-0857-1. DOI: 10.1145/2245276.2245357. URL: http://doi.acm.org/10.1145/2245276.2245357.

[3]   Alberto Bacchelli and Christian Bird. „Expectations, Outcomes, and Challenges of Modern Code Review". In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 712–721. ISBN: 978-1-4673-3076-3. URL: http://dl.acm.org/citation.cfm?id=2486788.2486882.

[4]   Loïck Briot, Pascal Urso, and Marc Shapiro. „High Responsiveness for Group Editing CRDTs". In: *ACM New York, NY, USA ©2016* (2016).

[5]   Bryden Cho, Agustina Ng, and Chengzheng Sun. „CoVim: Incorporating real-time collaboration capabilities into comprehensive text editors". In: *2017 IEEE 21st International Conference on Computer Supported Cooperative Work in Design (CSCWD)* (2017).

[6]   Jacek Czerwonka et al. „CodeFlow: Improving the Code Review Process at Microsoft". In: *Queue - Benchmarking* 16.5 (2018), p. 20.

[7]   S. Garg and S. Garg. „Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security". In: *2019 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*. 2019, pp. 467–470. DOI: 10.1109/MIPR.2019.00094.

[8]   Max Goldman, Greg Little, and Robert 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. Santa Barbara, California, USA: ACM, 2011, pp. 155–164. ISBN: 978-1-4503-0716-1. DOI: 10.1145/2047196.2047215. URL: http://doi.acm.org/10.1145/2047196.2047215.

[9]   Carl Gutwin, Reagan Penner, and Kevin Schneider. „Group Awareness in Distributed Software Development". In: *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*. CSCW '04. Chicago, Illinois, USA: ACM, 2004, pp. 72–81. ISBN: 1-58113-810-5. DOI: 10.1145/1031607.1031621. URL: http://doi.acm.org/10.1145/1031607.1031621.

[10]   Gertjan Halkes and Johan Pouwelse. „UDP NAT and Firewall Puncturing in the Wild". In: *NETWORKING 2011*. Ed. by Jordi Domingo-Pascual et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–12. ISBN: 978-3-642-20798-3.

[11]   J. D. Herbsleb and R. E. Grinter. „Architectures, coordination, and distance: Conway's law and beyond". In: *IEEE Software* 16.5 (1999), pp. 63–70. ISSN: 0740-7459. DOI: 10.1109/52.795103.

[12] Aditya Kurniawan, Christine Soesanto, and Joe Erik Carla Wijaya. „CodeR: Real-time Code Editor Application for Collaborative Programming". In: *Procedia Computer Science* 59 (2015), pp. 510–519.

[13] Magnus Madsen, Ondřej Lhoták, and Frank Tip. „A Model for Reasoning About JavaScript Promises". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 86:1–86:24. ISSN: 2475-1421. DOI: 10.1145/3133910. URL: http://doi.acm.org/10.1145/3133910.

[14] Gérald Oster et al. „Data consistency for P2P collaborative editing". In: *ACM New York, NY, USA ©2006* (2006).

[15] Gérald Oster et al. „Supporting String-Wise Operations and Selective Undo for Peer-to-Peer Group Editing". In: *ACM New York, NY, USA ©2006* (2014).

[16] M. Paasivaara, S. Durasiewicz, and C. Lassenius. „Distributed Agile Development: Using Scrum in a Large Project". In: *2008 IEEE International Conference on Global Software Engineering*. 2008, pp. 87–95. DOI: 10.1109/ICGSE.2008.38.

[17] M. Paasivaara, S. Durasiewicz, and C. Lassenius. „Using Scrum in Distributed Agile Development: A Multiple Case Study". In: *2009 Fourth IEEE International Conference on Global Software Engineering*. 2009, pp. 195–204. DOI: 10.1109/ICGSE.2009.27.

[18] Shaun Phillips, Jonathan Sillito, and Rob Walker. „Branching and merging: an investigation into current version control practices". In: *ACM New York, NY, USA ©2011* (2011), p. 12.

[19] Nuno Preguica et al. „A Commutative Replicated Data Type for Cooperative Editing". In: *IEEE* (2009).

[20] Jonathan Protzenko et al. „Implementing real-time collaboration in TouchDevelop using AST merges". In: *MobileDeLi 2015 Proceedings of the 3rd International Workshop on Mobile Development Lifecycle* (2015), pp. 25–27.

[21] Rajiv Ranjan and Liang Zhao. „Peer-to-peer service provisioning in cloud computing environments". In: *The Journal of Supercomputing* 65.1 (2013), pp. 154–184. ISSN: 1573-0484. DOI: 10.1007/s11227-011-0710-5. URL: https://doi.org/10.1007/s11227-011-0710-5.

[22] Rajiv Ranjan et al. „Peer-to-Peer Cloud Provisioning: Service Discovery and Load-Balancing". In: *Cloud Computing: Principles, Systems and Applications*. Ed. by Nick Antonopoulos and Lee Gillam. London: Springer London, 2010, pp. 195–217. ISBN: 978-1-84996-241-4. DOI: 10.1007/978-1-84996-241-4_12. URL: https://doi.org/10.1007/978-1-84996-241-4_12.

[23] Caitlin Sadowski et al. „Modern code review: a case study at google". In: *ICSE-SEIP '18 Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (2018), p. 186.

[24] Ken Schwaber. *Agile project management with Scrum*. Microsoft press, 2004.

[25] Ken Schwaber. „SCRUM Development Process". In: *Business Object Design and Implementation*. Ed. by Jeff Sutherland et al. London: Springer London, 1997, pp. 117–134. ISBN: 978-1-4471-0947-1.

[26] D. Spinellis. „Git". In: *IEEE Software* 29.3 (2012), pp. 100–101. ISSN: 0740-7459. DOI: 10.1109/MS.2012.61.

[27] Mohsen Nader Tehrani, Murat Uysal, and Halim Yanikomeroglu. „Device-to-device communication in 5G cellular networks: challenges solutions and future directions". In: *IEEE Communications Magazine* 52 (2014), pp. 87–88.

[28] P. Thongtanunam et al. „Who should review my code? A file location-based code-reviewer recommendation approach for Modern Code Review". In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2015, pp. 141–150. DOI: 10.1109/SANER.2015.7081824.

[29]   D. Widder et al. „I'm Leaving You, Travis: A Continuous Integration Breakup Story". In: *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 2018, pp. 165–169.

[30]   R Yuzuki, H Hata, and K Matsumoto. „How we resolve conflict: an empirical study of method-level conflict resolution". In: *Software Analytics (SWAN), 2015 IEEE 1st International Workshop on*. 2015, pp. 21–24. DOI: 10.1109/SWAN.2015.7070484.

A   Appendix

Listings, data models, forms, …