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: Dipl. Ing. Johann Grabner

Vienna, May 15, 2019

Kurzfassung

Über diese Vorlage: Dieses Template dient als Vorlage für die Erstellung einer wissenschaftlichen Arbeit am INSO. Individuelle Erweiterungen, Strukturanpassungen und Layout-Veränderungen können und sollen selbstverständlich nach persönlichem Ermessen und in Rücksprache mit Ihrem Betreuer vorgenommen werden.

Aufbau: In der Kurzfassung werden auf einer 3/4 bis maximal einer Seite die Kernaussagen der Diplomarbeit zusammengefasst. Dabei sollte zunächst die Motivation/der Kontext der vorliegenden Arbeit dargestellt werden, und dann kurz die Frage-/Problemstellung erläutert werden, max. 1 Absatz! Im nächsten Absatz auf die Methode/Verfahrensweise/das konkrete Fallbeispiel eingehen, mit deren Hilfe die Ergebnisse erzielt wurden. Im Zentrum der Kurzfassung stehen die zentralen eigenen Ergebnisse der Arbeit, die den Wert der vorliegenden wissenschaftlichen Arbeit ausmachen. Hier auch, wenn vorhanden, eigene Publikationen erwähnen.

Wichtig: Verständlichkeit! Die Kurzfassung soll für Leser verständlich sein, denen das Gebiet der Arbeit fremd ist. Deshalb Abkürzungen immer zuerst ausschreiben, in Klammer dazu die Erklärung: z.B: „Im Rahmen der vorliegenden Arbeit werden Non Governmental-Organisationen (NGOs) behandelt, ...“. In LaTeXwird diese bereits automatisch durch verwenden des Befehls `\ac` erreicht. Für Details siehe Paket `glossaries`.

Schlüsselwörter

Abstract

About this template: This template helps writing a scientific document at INSO. Users of this template are welcome to make individual modifications, extensions, and changes to layout and typography in accordance with their advisor.

Writing an abstract: The abstract summarizes the most important information within less than one page. Within the first paragraph, present the motivation and context for your work, followed by the specific aims. In the next paragraph, describe your methodology / approach, and / or the specific case you are working on. The third paragraph describes the results and the contribution of your work.

Comprehensibility: People with different backgrounds who are novel to your area of work should be able to understand the abstract. Therefore, acronyms should only be used after their full definition has given. E.g., "This work relates to non-governmental organizations (NGOs), . . . ".

Keywords

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

1.1 Problem Description

When multiple people are working on solving a Problem it is not clear who is implementing which part. Current vcs systems conflict resolutions require manual conflict resolution if multiple people have modified the same file. In order for a group to discuss a problem all the code has to be commited and pulled by everyone first. This introduces friction. In order to solve these and other problems real time collaborative editors have been created. But current implementations of real time collaboration editors / editor plugins are unaware of the underlying version control system and therefore are based on the idea of just sharing files of a host mashine or a single source of truth file on a server instead of basing edit histories on versions of files known to the version control system anyway.

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. In order to convert this concurrent model to a commit understood by Git it should be possible to stage changes by author. This approach might require more sophisticated logic than just interpreting changes as strings given that 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.

1.2 Expected Results

Enable real time collaboration on source code based on a Git[1] project by continuous tracking of code changes synchronizing over peer to peer connection. Lowering overhead of splitting tasks by enabling everyone see what other people are working on. Allowing discussions about source code that has not yet been commited. Changes should be commitable by author. In order to be able to compile sourcecode, 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 will be propagated to other people working on the same branch. If not the changes will be sent when a connection is available. [1] Therefore it should be analog to the benefits of moving to a decentralized vcs.

1.3 Motivation

Current implementations of real time collaboration tools are not designed with version control systems in mind.

Transforming real time collaborative edits into regular Git commits by author will reduce a lot of friction in the adoption for real time collaboration software. Using a peer to peer solution with the ability to deal with disconnect events using information already known to the version control system will drasticly increase the ease of use. Ideally a user will not even have to think about using the extension.

[1] https://git-scm.com/

2 Fundamentals

2.1 State of the Art

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

2.1.1 Teletype for Atom

Teletype for Atom[1] is a Project enabling editing files peer to peer. It is based on [5] [6] [2]. 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 they are opened by the host.

2.1.2 CoVim

CoVim[3] uses Operational Transforms just like Teletype for Atom. But uses a different method of detecting state changes. Instead of observing user interactions it observes changes to files and generates Operational Transforms from diffing states.

2.1.3 TouchDevelop

TouchDevelop[3] is an experimental web based editor. It enables real time collaboration by merging ASTs. The paper claims "Indeed, we claim that this approach is generalizable to a general-purpose language, as long as the editor can parse the program being edited and transparently tag AST nodes with an identifier. Naturally, this requires non-trivial support from the editor".[9] The AST translation would have to be implemented for every programming language supported. Additionally the transformation from AST to text might not guarantee the same code just the same AST which could lead to confusion for developers.

2.1.4 CodeR

CodeR[4] is a Web IDE with built in chat for the programming languages C, C++ and JAVA.

2.1.5 Visual Studio Live Share

Visual Studio Live Share[4] 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 ports opened by debugging sessions are forwarded to clients. As with Teletype for Atom files are only persisted on the "host"

[1] https://github.com/atom/teletype/issues/211
[2] https://teletype.atom.io/
[3] https://www.touchdevelop.com
[4] https://visualstudio.microsoft.com/de/services/live-share/

Tool	Type	Location	Shared Content
Teletype for Atom	Extension	Host	Individual Files
Visual Studio Live Share	Extension	Host	Project Folder
Multihack Brackets	Extension	Distributed?	Project Folder
Codeshare	Web application	Server	Single File

Table 21: Overview state of the art

2.1.6 Multihack-Brackets

Multihack-Brackets[5] is a plugin for the Brackets 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 just crashes the brackets editor.

2.1.7 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.

2.1.8 Summary

As described in 21 current solutions are mostly implemented as extensions to code editors. Only codeshare is web based because it is based around the use-case of doing interviews. Overall none of the current solutions have any considerations for dealing with an underlying version control system of a project.

2.2 CRDT

In order to update shared objects stored at different sites a "commutative replicated data type" or CRDT is proposed by [8]. 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. But 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 be established. [8],[5] have a solution to this problem: every site gets a unique siteID 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 the siteID.

2.3 Code Review

Companies like Google use a process called "code review" to validate changes. The basic idea is that before merging a code modification onto master, it is reviewed by someone else. This keeps code quality stable and educates developers.[10] This thesis aims to support these workflows while enhancing the developer experience.

[5] https://github.com/multihack/multihack-brackets
[6] https://codeshare.io

3 Methodological Approach

First a state of the art analysis will be conducted. This analysis will look into currently available open-source and commercial products for real time collaboration. Additionally the current literature related to real time collaboration will be studied. The papers mentioned in open source projects will be used as a starting point for forwards and backwards literature analysis.

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

The literature survey will be continuously documented in the following format:

```
<date>: <search query>
  <publisher>
    <paper title>
<search query> := one of the following:
```

- search terms
- conference
- forward/backward search paper title
- authors of specific papers

If multiple searches occur 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 will be defined. These requirements will be user centered. Functional as well as non functional aspects such as performance will be defined as requirements.

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

- Actor (the steakholder)
- 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 accomodate the identified use-cases will be implemented. The extension will meet the following requirements:

- The extension will target VS Code 1.32 or newer and will be written in node JS
- Automated testing will be used wherever possible and meaningful.
- The source code will be commented and well documented.

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

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

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

4 Requirements

4.1 Share the entire project structure

Every user should be able to see all changes to the project. Based on this Issue and the discussion "Current implementation only shares current project. It would be more useful if I could share the whole project so two persons can work on different files."[1] being able to edit the entire project structure is a very important feature.

4.2 Only display changes in the same Git branch

Given that a lot of developers are using prototype branches and a significant number are usign feature branches Git branches are a good indication that a specific problem is being worked on.[7] Therefore only displaying cuncurrent edits on the same edit removes noise of unrelated edits.

4.3 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 modicications would be bundled into huge commits. In order to mitigate this, changes should be stageable by author.

4.4 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 information (such as the current username) from Git.

4.5 Respect .gitignore

Files explicitly excluded from the version control system via the .gitignore file[3] should not be synchronized with other client as these files might contain automatically generated files that depend on the local system configuration or contain sensitive information.

4.6 Performance

Although "VS Code aims to deliver a stable and performant editor to end users, and misbehaving extensions should not impact the user experience. The Extension Host in VS Code prevents extensions from:

[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

- Impacting startup performance
- Slowing down UI operations
- Modifying the UI

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

Scenario: A person is sitting in front of two computers both with vscode 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.

4.7 Disable foreign changes

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

4.8 Support bad internet connection

The extension should be able to support the client loosing the internet connection. Even if the code editor is relaunched 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] https://code.visualstudio.com/api/advanced-topics/extension-host

5 User stories

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

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

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

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

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

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

As a programmer I want to be able to edit files with up to 4 people at a time. A file can have up to 30.000 characters.

6 Implementation

6.1 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.

It provides functions to:

- Find the Git directory for a given file (given the file is inside a Git repository)
- Get the current branch of a Git repository
- Get the remote URL for a repository
- Get the path of a file within a repository
- Get a list of commit hashes
- Get the current version of a given file
- Stage a file for commit
- Get the username configured for the Git repository
- Get a list of branches of a Git repository
- Do a Git reset
- Switch a repository to a different branch

6.2 teletype-crdt

"The string-wise sequence CRDT powering peer-to-peer collaborative editing in Teletype for Atom."[1] This library will be used for tracking changes. It is written in JavaScript and currently does not include an API documentation.

The main functions used are:

[1] https://github.com/atom/teletype-crdt

- 6.1 to notify the document representation about a text operation on the local document
- 6.2 to notify the document representation about remote changes and returns the changes to the local file required to keep it in sync with other instances
- 6.3 to generate diffs for hiding remote changes as well as staging changes

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

Listing 6.1: teletype-crdt setTextInRange

```
1 integrateOperations(operations: [operations]): {textUpdates:[
  ↪ textUpdate], markerUpdates:any};
```

Listing 6.2: teletype-crdt integrateOperations

```
1 undoOrRedoOperations(operationsToUndo: [operation]): any;
```

Listing 6.3: teletype-crdt undoOrRedoOperations

6.2.1 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.

6.3 VS Code Extension API

VS Code runs extension in a seperate process and provides an asynchronous Javascript API. The examples provided in the vscode-extension-samples repository[2] are mostly written in Typescript[3] and all the Interfaces have type definitions for Typescript. Therefore the Extension will use Typescript as well.

6.3.1 Used API Functions

The VS Code API provides a set of asynchronous functions that return Promises. The extension uses the following set:

- 6.4 "An event that is emitted when a text document is changed. This usually happens when the contents changes but also when other things like the dirty-state changes."[4] to detect user modifications to files
- 6.5 "An event that is emitted when a text document is opened or when the language id of a text document has been changed."[5] to initialize the document representation when a user opens a file

[2] https://github.com/Microsoft/vscode-extension-samples
[3] https://www.typescriptlang.org/
[4] https://code.visualstudio.com/api/references/vscode-api#workspace
[5] https://code.visualstudio.com/api/references/vscode-api#workspace

- 6.6 "Make changes to one or many resources or create, delete, and rename resources as defined by the given workspace edit."[6] to apply edits from other peers to a file

```
1 vscode.workspace.onDidChangeTextDocument()
```

Listing 6.4: VS Code API onDidChangeTextDocument

```
1 vscode.workspace.onDidOpenTextDocument()
```

Listing 6.5: VS Code API onDidOpenTextDocument

```
1 vscode.workspace.applyEdit(edit)
```

Listing 6.6: VS Code API applyEdit

6.6 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 noticable when changes by regular typing are integrated into the local document. But when the multiple cursors[7] are used the change integration can slow down noticably. In order to support multiple cursors the changes are sorted by column in decending order to prevent shifting indices. (See 6.7)

```
1 textUpdates.sort((a, b) => b.oldStart.column - a.oldStart.column)
```

Listing 6.7: Sorting Changes by Column to Prevent Index Shifting

6.3.2 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. 61

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

```
"contributes": {
    "viewsContainers": {
        "activitybar": [
            {
                "id": "change-explorer",
                "title": "Change Explorer",
                "icon": "media/icon.svg"
            }
        ]
    }
    [...]
}
```

Listing 6.8: Tree View Activitybar

First an entry in the activity bar has to be declared in the viewsContainers section (See 6.8). It defines the icon as well as the hover text (called title) of the tab in the activity bar.

```
"contributes": {
    [...]
    "views": {
        "change-explorer": [
            {
                "id": "contributors",
                "name": "Contributors"
            }
        ]
    }
}
```

Listing 6.9: Tree View Pannel Definition

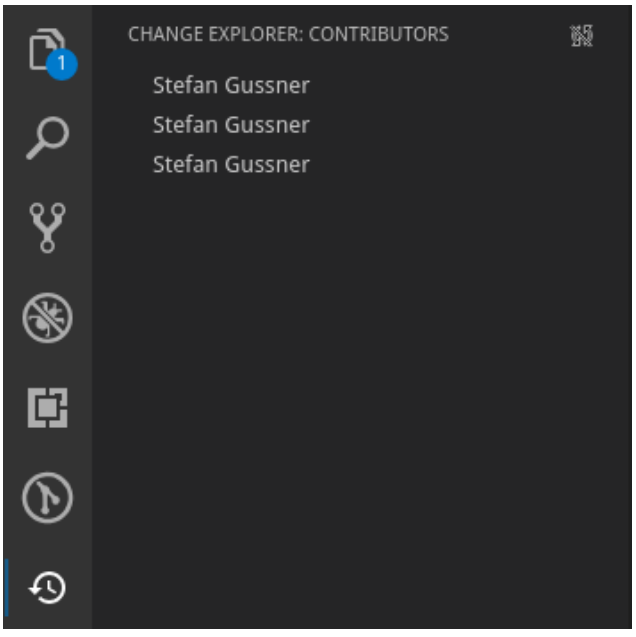

Figure 61: Tree view

Additionally the view has to be declared.6.9 This defines the heading for the Tree View.

```
1 const treeview = new ContributorsTreeView(crdt.getUsers);
2 vscode.window.registerTreeDataProvider('contributors', treeview);
```

Listing 6.10: Define Tree View Data Provider

To populate the Tree View with data, a TreeDataProvider has to be registered to the view id. (See 6.10) The TreeDataProvider interface defines functions that return all items for the Tree View as well as refresh the content of the Tree View.

6.4 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 6.11: Data Model Declarations

All the data is stored in a map called documents.(See 6.11) 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 filepath.

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

6.4.1 The Document Object

The document object has two properties

- document
- metaData

The document property points to an instalce 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 usersnames in the staging Tree View.

6.5 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 6.12) 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.filename == e.document.fileName);
12 if (knownChanges.length > 0) {
13     //remove from known changes
14     currentChanges.splice(currentChanges.indexOf(knownChanges[0]));
15 }
```

Listing 6.12: 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 6.13)

```
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 6.13: Network Data Packet

6.6 Network Transport

6.6.1 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 as well as the port of the socket, the peer is listening on for incoming connections. Upon recieving this information the bootstrap server responds with a list of ips and ports of all the other registered peers. The peer then tries to connect to every peer in that list.

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

6.6.2 Handling Changes

Every change is propergated 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 propergated via a gossip based protocol.

6.7 Handling Remote Changes

Handling incoming changes has two main challanges:

- concurrent changes
- finding files on disk

6.7.1 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 it's index changed by the other one.

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. 6.14 ensures that incoming change packets are processed sequentially by building a promise chain.

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

Listing 6.14: Network Promise Chain

6.7.2 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.

6.7.3 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 6.15)

```
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 6.15: Adding Change to Local Document

7 Ergebnisse

Die Resultate der Arbeit präsentieren und nach Möglichkeit aussagekräftige, eigenständige Abbildungen einbauen. Namen des Kapitels konkretisieren, an jeweilige Arbeit anpassen – Lösungsvorschlag/Implementierung im Titel des Kapitels benennen.

8 Conclusions

Procurrently demonstrates an approach to real time collaboration compatible with traditional workflows 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 commiting to a Git repository by staging changes by author.

8.1 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 [11]
- Encrypt traffic and authenticate collaborators

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

[3] 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).

[4] 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.

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

[6] 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).

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

A Appendix

Listings, data models, forms, ...