Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

MURAD ISAYEV
MASTER THESIS

CONFLICT RESOLUTION IN COLLABORATIVE WEB APPS WITH OFFLINE USE

Submitted on 31 March 2019

Supervisor: Prof. Dr. Dirk Riehle, M.B.A., Andreas Kaufmann, M.Sc.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 31 March 2019

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see https://creativecommons.org/licenses/by/4.0/

Erlangen, 31 March 2019

Abstract

Offline usage is a great advantage for the web apps. It gives the opportunity to interact with the app without the need of constantly having the network connection. After network recovery the app synchronizes the collected offline data with the backend data. However, when the synchronized data is shared with other users, it may lead to possible data loss. The last synchronized user overwrites the backend resource with its own version. Thus, the user must be notified about the conflict before posting his version into backend. Additionally the user must be provided with the opportunity to resolve the arisen conflict. This is highly likely to happen in collaborative web apps, where many users work together on a single project. This thesis examines different version control / merging methods and provides a possible solution with the implementation of such methods at the example of QDACity application.

Contents

1 Introduction **1**
1.1 Collaborative Software . . . 2
1.2 Progressive Web Application . . . 2
1.2.1 Service Worker . . . 2
1.2.2 IndexedDB . . . 3
1.3 Three-Way Merge . . . 3
1.4 Edit Distance . . . 5
1.4.1 Levenshtein distance . . . 6
1.4.2 Damerau–Levenshtein distance . . . 6
1.4.3 Longest Common Subsequence . . . 7
1.4.4 Hamming distance . . . 7
1.4.5 Jaro–Winkler distance . . . 7
1.4.6 Discussion of presented metrics . . . 7

2 Requirements **9**
2.1 Functional Requirements . . . 9
2.2 Non-functional requirements . . . 9

3 Architecture and Design **10**

4 Implementation **11**
4.1 Handlers . . . 11
4.1.1 SyncHandler . . . 11
4.1.2 CodeDiffHandler . . . 11
4.1.3 TextDiffHandler . . . 11
4.2 App Components . . . 11
4.2.1 Code Conflict . . . 11
4.2.2 Text Document Conflict . . . 11
4.3 Tests . . . 11

5 Evaluation **12**
5.1 Functional Requirements . . . 12

5.2 Non-functional requirements 12

References **13**

List of Figures

1.1 Automatic merge 4
1.2 Manual merge 4
1.3 Auto and manual merge 5

1 Introduction

This thesis is focused on the conflict resolution problem for the web application QDAcity built by Professorship Open Source University Erlangen-Nuremberg. QDAcity is a quantitative analysis tool, which allows researchers create their own projects and perform analysis on them. One of the major features of QDAcity is *real time collaborative editing*. It gives the opportunity to the users to work on the same data simultaneously. Applications main entity is *Project*. *Project* collects all the results of the work done by the researches. Interaction with the *Project* is performed through application component *Coding Editor*. This component includes tools for creating project artifacts, such as *Code* and *Text Document*. *Coding Editor* also includes text editor utility to edit the created text documents. *Code* and *Text Document*, however, are the main subject of the thesis topic, as all the conflict scenarios described in the following chapters are based on them. Whereas *Text Document* represents an object storing actual text value of the document, *Code* is an data structure containing some descriptive attributes such as *Name*, *Author*, *Color*, *Memo* etc.

Another major feature of QDAcity is the *working offline* functionality. It means, that users don't necessarily need to have the network connection to be able to perform operations on the project resources. However, this feature creates potential complication. Since the user has no access to the original data stored on the server, he executes his work on the copy of the data which is locally stored on his browser. Data in this particular example represents *Code* and *Text Document* objects. Offline user can manipulate *Code* by changing any of its property values, e.g. *Name*. Additionally he may also perform edit operations on the *Text Document* by adding new text snippets or removing some parts of its value. As a result of such actions, user creates his own version of the edited objects stored locally. After network recovery his data version must be uploaded to the server. However, if the original version of the edited offline data was also updated by someone else, potential conflict situation arises. The thesis's aim is to handle detection of such conflicting situations. Moreover, automatic merging of non-conflicting changes is also part of the thesis problem. As a final result, the users of QDAcity application shall be provided with an UI that notifies them about

arisen conflicts. Furthermore, UI must be supplied with the tool displaying differences between versions. The tool is also expected to allow the user to resolve the conflict by selectively picking desired version.

Chapter 1 gives some background to the subtopics derived from the thesis topic, as well as the theoretical base for the implemented solutions to the thesis problem. Purpose of the project and requirements are defined in the Chapter 2. Chapter 3 presents the architecture and design of the related software components, as well as, the overview of the implemented algorithms. Implementation of selected methods is described in Chapter 4. Finally Chapter 5 describes the evaluation of fulfilment of the presented requirements.

1.1 Collaborative Software

Collaborative Software is an application software which allows to several users to work on common project to accomplish the common goals. Level of collaborative interaction is classified into following types:

- *Real-Time Collaborative Editing* is a form of user interaction with collaborative application when multiple users are able to edit the shared data simultaneously.
- *Version Control* stands for the collaboration behaviour when each user has his own version of copied shared data after performing edit operations on it.

1.2 Progressive Web Application

Over the years Web and Desktop applications had inverse set of advantages and disadvantages in relation to each other. Desktop applications deliver high performance and network status independence. However, they require storage costs and lack real-time update opportunities. Web apps eliminate inconveniences related to desktop apps, but with bottleneck point of network dependency. Consequently, *Progressive Web Application* is the solution that incorporates advantages of both application types, which provides *working offline* functionality, running in turn in a web browser. Progressive Web Application technologies that are used in QDAcity app are described in the following subsections.

1.2.1 Service Worker

Service Worker is a network proxy between browser and web-server. It takes the HTTP requests and depending on network status performs the handling of received requests. For instance, if the browser has the network connection, the

accepted HTTP request can directly be redirected to the web-server. In case of offline experience, the requests are cached and later on sent to the server after network recovery.

1.2.2 IndexedDB

IndexedDB is a data storage which is embedded in web browser. The usage of IndexedDB is especially advantageous for progressive web applications (see section 1.2) and QDAcity in particular, since it allows to directly perform CRUD[1] operation on the application data independently on network condition.

1.3 Three-Way Merge

Three-Way Merge is a technique used to automate the synchronization process. Parallel editing of copies and further integration of this modification into merged copy is one of the crucial aspects of collaborative work (Balasubramaniam and Pierce, 1998). It assumes considering original version, after performing difference search between data versions A and B. If data in both versions were changed in different areas (e.g, lines), then the synchronization should automatically merge this changes without user involvement. In case of changes made in the same fields, conflicting areas must be recognized and flagged.

Three-Way Merge can run into 3 possible scenarios: *Automatic Merge* and *Manual Merge* and combination of both.
Example of automatic merge:

[1] Create Read Update Delete

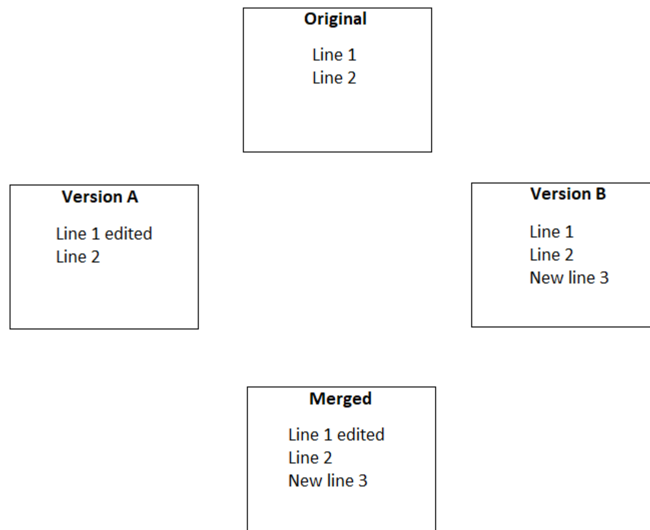

Figure 1.1: Automatic merge

In this example Line 1 was edited in Version A and a New Line 3 was added in the Version B after second line. Since both separate operations were performed on different lines, they don't run into a conflict. Therefore all modifications are automatically integrated into merged version.

Example of manual merge:

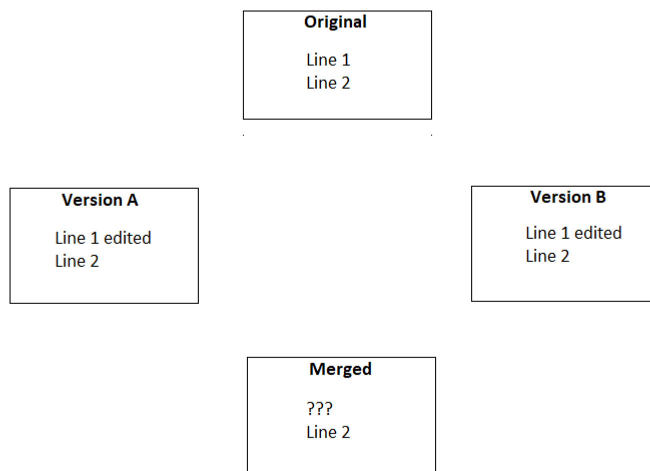

Figure 1.2: Manual merge

This is the example when both versions were modified at the same line. As a result, auto merge is not possible in this case, since it is ambiguous how conflicting

line should look like in the merged version. Thus, user interaction is needed to perform manual merge.

Example of auto and manual merge:

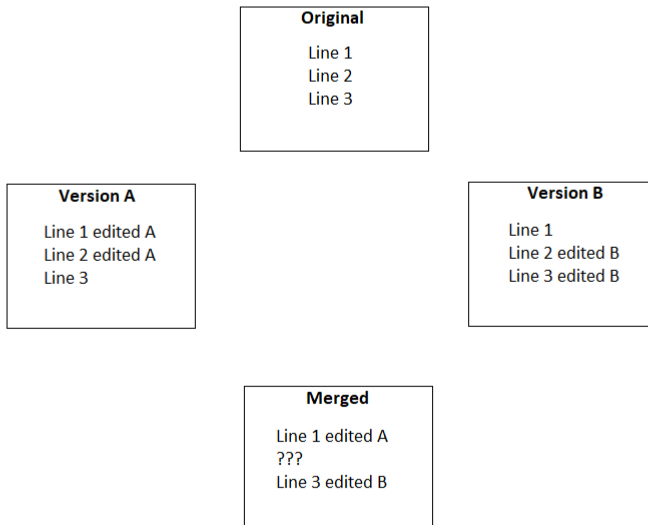

Figure 1.3: Auto and manual merge

In this example Line 2 were modified in both versions, whereas only Line 1 and only Line 3 were edited in Version A and Version B respectively. Obviously both versions are conflicting, but before the manual merging, non-conflicting parts of files must be automatically merged.

Matching above mentioned examples with similar cases that may appear in QDAcity app, Code attributes and TextDocument paragraphs can be considered as separate lines used in this examples. Hence same merging approaches can be applied when Code attributes or TextDocument paragraphs get modified. However, there is one extension point related to TextDocument conflict case. Besides conflicting paragraphs detection, there is a need to determine the parts of the paragraphs' inner content where they differ. In order to fulfil this requirement string difference metrics need to be considered, which are described in detail in the following section.

1.4 Edit Distance

Edit Distance, also referred to as *Minimum Edit Distance* is a minimum number of edit operations to be performed to convert one string into another (Ristad and

Yianilos, 1997). There are 4 types of edit operations that can be determined in a string difference analysis (Navarro, 1999):

- *Insertion* is an operation the result of which produces new string with inserted character. E.g. inserting *a* to the string *ab* produces *abc*.
- *Deletion* of an character from the original string. E.g. deletion of *c* from *abc* $\rightarrow$ *ab*.
- *Substitution* is a replacement of a single character at specific position with another one. E.g. substitution of *b* in *abc* with *x* $\rightarrow$ *axc*.
- *Transposition* is an operation of swapping two adjacent characters in a single string. E.g. transposition of *b* and *c* in *abcd* $\rightarrow$ *acbd*.

There are quite a few approaches used to define edit distance and most popular of them are presented in the next subsections. Finally the chapter closes with the discussion of this approaches and selection the most suitable one for the solution of the thesis problem.

1.4.1 Levenshtein distance

Levenshtein distance (Levenshtein, 1965) considers using 3 of the mentioned in Chapter 1.4 operations. Basically it allows any number *insertion*, *deletion* and *substitution* operations to transform a string to another one. The main condition however for the resulting number is that it should be minimum. The definition can also be rephrased as *"the minimal number of insertions, deletions and substitutions to make two strings equal."* (Navarro, 1999) Let the strings be defined as *x*, *y* and their length as $|x|$, $|y|$ then the resulting distance holds: $0 \leq d(x,y) \leq max(|x|, |y|)$. Consequently, distance equals to 0 when the strings are equal, and in its upper bound the distance is the length of the longest string.

1.4.2 Damerau–Levenshtein distance

This string metric also involves three classical edit operations (insertion, deletion and substitution). However, it extends *Levenshtein distance* (See 1.4.1) by introducing additional *transposition* operation. It is widely used in spell checking programs, as it was initially designed to calculate the distance between spelling errors (Peterson, 1980). Damerau (1964) points out that more than 80% of human misspellings caused by: 1) accidental *transposition* (swapping) two adjacent letters 2) *inserting* one extra letter by accident 3) missing a letter in the word (*deletion*) 4) wrong letter typing (*substitution*). Function's upper and lower bounds are the same as in the *Levenshtein distance* (See 1.4.1).

1.4.3 Longest Common Subsequence

Longest Common Subsequence is another string metric which calculates the length of the subsequence of the characters that present in both strings. It examines however only *insertion* and *deletion* operations. It is used as a base for the algorithm of Unix *diff* and *diff3* tools (GNU, n.d.) (Pierce, Khanna, Kunal & Pierce, 2016). Let the strings be defined as x, y and their length as $|x|$, $|y|$ then the distance holds: $0 \leq d(x,y) \leq |x| + |y|$ (Navarro, 1999). Since it doesn't support *substitution* operation, in comparison with *Levenshtein distance* (See 1.4.1) its highest possible value equals to sum of the length of both strings (that is the case when all characters in strings differ).

1.4.4 Hamming distance

Hamming distance counts the positions in two strings of *equal length* where the matching characters differ. Given that this metric is limited to the condition of having same length for the analysed strings, it doesn't allow *insertion* and *deletion* operations. In other words, it is the numbers of *substitutions* required to alter one string into another. Therefore, distance holds: $0 \leq d(x,y) \leq |x|$ (Navarro, 1999).

1.4.5 Jaro–Winkler distance

Jaro–Winkler distance is a string comparator metric which comparative feature is allowance only *transposition* operations. It was presented by Winkler (1990) as an alternative to *Jaro distance* metric (Jaro, 1989). It describes two strings similarities by the number 0 and 1, where 1 means exact match of both strings (Winkler, 1990). However it is best suitable for short strings, as it was designed to analyse short character sequences (Cohen, Ravikumar & Fienberg, 2003).

1.4.6 Discussion of presented metrics

Although described in the previous five metrics are used to determine the differing *characters* in strings, their basic concept can easily be applied also for detection of differing words in the paragraph. This can be implemented by simply assuming single characters used in the examples in Chapter 1.4 as words (i.e. character sequences separated by *whitespace character*). The main rules of *edit distance* are still applicable in this case.

Examining the presented string metrics list, *Hamming distance* (see 1.4.4) is obviously not suitable one for the solution of the thesis problem. The reason for that conclusion is that, this metric requires analysed strings to be of same length and *TextDocument* paragraphs values are not limited in length. Usage of *Longest Common Subsequence* see(1.4.3) could be a good decision indeed,

furthermore it is been already used in different *diff* tools. However, in comparison with *Levenshtein* (see 1.4.1) and *Damerau–Levenshtein* (see 1.4.2) distances, it lacks supporting one major edit operation type which is crucial for the problem - *substitution*. That is, displaying replaced words in the conflict visualization is one of the requirements for this thesis. Although *Damerau–Levenshtein distance* is an improved version of *Levenshtein distance*, its distinctive advantage is not relevant for the topic: detection of *transposed* words is not what conflict tool needs. This is the reason why *Jaro-Winkler distance* (see 1.4.5) is also not considered as a solution for QDAcity application.

Taking into consideration advantages and disadvantages of the presented metrics, as well as degree of their compatibility with the thesis requirements, *Levenshtein distance* is decided to be used as a base for *TextDocument* paragraphs diff algorithm. Implementation of such algorithm will help to determine which words were *inserted* to the updated TextDocument paragraph. Additionally, words *deleted* from original documents, as well as replaced (*substituted*) words can be flagged using this algorithm.

2 Requirements

The objective of thesis is designing and implementation version control / conflict resolution use case of existing synchronization process of QDAcity application. This solution considers only the objects that can be modified in offline mode in the current version of the application. Those objects in particular are *Code* and *TextDocument*, that were described in Chapter 1. The further detailed description of all functional and non-functional requirements is presented in the following sections of this chapter.

2.1 Functional Requirements

2.2 Non-functional requirements

3 Architecture and Design

4 Implementation

4.1 Handlers

4.1.1 SyncHandler

4.1.2 CodeDiffHandler

4.1.3 TextDiffHandler

4.2 App Components

4.2.1 Code Conflict

4.2.2 Text Document Conflict

4.3 Tests

5 Evaluation

5.1 Functional Requirements

5.2 Non-functional requirements

References

Balasubramaniam, S. & Pierce, B. C. (1998). What is a file synchronizer? In *Proceedings of the 4th annual acm/ieee international conference on mobile computing and networking* (pp. 98–108). MobiCom '98. doi:10.1145/288235.288261

Cohen, W. W., Ravikumar, P. & Fienberg, S. E. (2003). A comparison of string distance metrics for name-matching tasks. (pp. 73–78).

Damerau, F. (1964). A technique for computer detection and correction of spelling errors. *Commun. ACM*, *7*(3), 171–176.

GNU. (n.d.). Gnu diffutils. Retrieved February 16, 2019, from https://www.gnu.org/software/diffutils/

Jaro, M. A. (1989). Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, *84*(406), 414–420. doi:10.1080/01621459.1989.10478785

Levenshtein, V. I. (1965). Binary codes capable of correcting deletions, insertions and reversals. *Doklady Akademii Nauk SSSR*, *163*(4), 845–848. Original in Russian – translation in Soviet Physics Doklady 10(8):707-710, 1966.

Navarro, G. (1999). A guided tour to approximate string matching. *ACM COMPUTING SURVEYS*, *33*, 2001.

Peterson, J. L. (1980). Computer programs for detecting and correcting spelling errors. *Commun. ACM*, *23*(12), 676–687. doi:10.1145/359038.359041

Pierce, B. C., Khanna, S., Kunal, K. & Pierce, B. C. (2016). A formal investigation of diff3.

Ristad, E. S. & Yianilos, P. N. (1997). Learning string edit distance.

Winkler, W. E. (1990). String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In *Proceedings of the section on survey research* (pp. 354–359). Wachington, DC.