Study of Conflicts in Collaborative Editing

Hoai Le Nguyen

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Étude des conflits dans l'édition collaborative

ΔΔΔ

Study of Conflicts in Collaborative Editing

THÈSE

présentée et soutenue publiquement le 14 Janvier 2021

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Hoai-Le NGUYEN

Composition du jury

Président :	Horatiu CIRSTEA	Professeur, Université de Lorraine
Rapporteurs :	Sophie CHABRIDON	Professeure, Télécom SudParis
	Stefano ZACCHIROLI	Maître de conférence, Université de Paris
Examinateur :	Hala SKAF-MOLLI	Maître de conférence, Université de Nantes
Directeurs de thèse :	François CHAROY	Professeur, Université de Lorraine
	Claudia-Lavinia IGNAT	Chargée de recherche, Inria Nancy - Grand Est

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Acknowledgment

First of all, I would like to thank my supervisors, Mr Francois Charoy, Professor of the *Université de Lorraine* and Mrs Claudia-Lavinia Ignat, Research Scientist at Inria, Nancy - Grand Est for their guidance, great support and kind advise throughout my PhD thesis. This PhD study would not have been possible without their insightful feedback, patient support and encouragements.

I would like to acknowledge my colleagues, Quentin Laporte-Chabasse and Mr Gérald Oster for their valuable help for the collection of the ShareLaTeX logs. And I would like to thank all members of the COAST team for the great working atmosphere that I, as a foreigner, have during my PhD in Inria, Nancy - Grand Est.

I would like to acknowledge the funding received toward my PhD from *Project 911 - Vietnam International Education Department Fellowships* and *Project OpenPaaS*.

Finally, I would like to thank my family for their endless love, help and support.

Contents

List of Figures

List of Tables

Introduction (Français)

Contexte d'étude

L'édition collaborative a longtemps attiré l'attention des chercheurs du *'Computer-Supported Cooperative Work' (CSCW)*. De nombreuses recherches ont été menées, de nombreux outils ont été construits pour soutenir et améliorer l'édition collaborative. La plupart des recherches dans les années 1990 et au début de 2000 se concentrent sur la description des caractéristiques de l'EC et de ses avantages/inconvénients [1, 3, 4, 5], sur la classification des modes de travail [6, 4], sur la construction de la taxonomie l'EC [7, 8, 3, 4, 5] et sur la proposition d'exigences pour les outils prenant en charge l'édition collaborative (l'EC) [3, 9, 10]. Ils étaient tous basés sur des entretiens avec des personnes qui avaient participé à certains projets d'édition collaboratifs. Un travail notable issu de ses recherches a été proposé par Posner et al [3] dans lequel les auteurs présentent une taxonomie de l'EC en termes de quatre composantes : 'role', 'activity', 'document control method' et 'writing strategy'. Chacun d'eux offre une perspective différente de l'EC. Cette taxonomie a été étendue par Lowry et al [4] avec quelques raffinements et l'ajout du composant de *'Work Mode'* qui décrit la distance physique entre les membres du groupe (c'est-à-dire même emplacement ou emplacement différent) et la synchronisation des activités d'édition (c'est-à-dire synchrone ou édition asynchrone). Le composant *'Work Mode'* a également été présenté précédemment par Ellis et al [7] comme *'CSCW Matrix'* (ou *'Time Space taxonomy'*) comprenant quatre catégories : *[same place, same time], [same place, different time], [different place, same time] et [different place, different time].*

La *'CSCW matrix'*, suppose cependant que tous les membres du groupe utilisent le même mode de travail (asynchrone ou synchrone) pendant toute la collaboration. Ce n'est pas ainsi que les gens travaillent ensemble. Minor et al [6] ont montré que les gens utilisent différentes stratégies d'écriture lors de l'édition collective d'un document ou d'un programme. Par exemple, lorsque deux utilisateurs ou plus éditent un article ensemble, ils ne l'éditent pas simultanément tout le temps. Un utilisateur peut arrêter l'édition pendant un certain temps (un jour par exemple) et se reconnecter plus tard pour l'édition. De même, Posner et al [3] ont confirmé que les stratégies synchrone et asynchrone sont utilisées dans différentes phases du projet l'EC et ont suggéré que les systèmes l'EC doivent prendre en charge une transition en douceur entre ces deux modes. Au-delà des modes de travail standard 'asynchrone' et 'synchrone' qui considèrent le travail collaboratif comme un seul flux d'activité, Dourish et al [11] ont présenté le mode de travail 'multi-synchrone' qui considère les travaux collaboratifs comme *'multiple, parallel streams of activity'*. En mode de travail 'multi-synchrone', les participants travaillent de manière isolée tandis que les activités de travail se déroulent en parallèle (divergence). Leur travail individuel sera intégré (synchronisation) périodiquement. Si l'intervalle de temps entre les synchronisations est suffisamment petit par rapport aux actions d'édition des utilisateurs, tous les collaborateurs peuvent voir les activités d'édition des autres après un temps très court. Dans ce cas, le mode de travail 'multi-synchrone' est très proche du mode de travail 'synchrone' standard. Et dans le

cas contraire, si la synchronisation a lieu moins fréquemment (après des heures, des jours ou des semaines), le 'multi-synchrone' est similaire au mode de travail traditionnel 'asynchrone'. D'une autre manière, on peut considérer les modes 'asynchrone' et 'synchrone' comme deux aspects du mode 'multi-synchrone' avec une période de synchronisation différente.

Un conflit peut survenir lors de la phase d'intégration car les modifications apportées par les utilisateurs pendant la phase de divergence peuvent se chevaucher. Plus la période de divergence est longue, les conflits sont plus susceptibles de se produire. Des conflits peuvent survenir au niveau textuel ou sémantique. Le conflit textuel est le résultat de la fusion textuelle des changements simultanés de deux individus différents qui se trouvent, selon le contexte, sur la même ligne ou sur deux lignes adjacentes d'un fichier partagé. Les conflits textuels peuvent être détectés par la plupart des systèmes prenant en charge l'EC. Le conflit sémantique (également connu sous le nom de conflit d'ordre supérieur ou direct) est le conflit au niveau sémantique. Par exemple, dans le cas d'un développement logiciel basé sur Git, les modifications simultanées de deux développeurs sont fusionnées avec succès (c'est-à-dire sans conflit textuel). Cependant, si le logiciel ne peut pas être compilé ou obtient des échecs de test, l'intégration de ces changements simultanés est considérée comme un conflit sémantique.

Les conflits sont coûteux car ils retardent le processus de développement. Elle appelle à une meilleure compréhension de la fréquence et du moment où les conflits sont plus susceptibles de se produire lorsque des personnes travaillent ensemble, en particulier en mode de travail 'asynchrone'. Plusieurs études avaient été menées, axées sur les changements parallèles et les conflits dans différents systèmes soutenant la collaboration tels que [12] (Ficus filesystem), [13] (Lucent Technologies' 5ESS), [14] (CVS) , [15, 16] (Git). Tous ces systèmes prennent en charge le suivi des changements d'intégration ('merging'), c'est un corpus précieux à analyser en plus des données basées sur les entretiens. Parmi eux, Git [17] - un système de contrôle de version décentralisé est devenu populaire dans la communauté de développement logiciel vers 2005. Alors que les systèmes de contrôle de version centralisés tels que CVS [18] ou Subversion [19] reposent sur un client-serveur, Git repose sur l'architecture peer-to-peer. Dans cette architecture, chaque client conserve une copie complète ('clone') du projet partagé. Les utilisateurs peuvent travailler de manière isolée sur leur référentiel local et peuvent synchroniser leurs travaux avec ceux des autres collaborateurs. Une 'fusion' est effectuée lorsque les utilisateurs choisissent de se synchroniser. Git, de facon similaire à d'autres systèmes de contrôle de version, utilise une technique de fusion textuelle [20] dans laquelle les 'lignes'sont considérées comme des unités indivisibles. Lorsqu'un fichier est édité en parallèle, nous appelons les modifications effectuées par différents collaborateurs *'conflicting changes'*. Si les *'conflicting changes'* font référence à la même ligne ou aux lignes adjacentes du fichier, alors ils forment un *'unresolved conflicts'* qui ne peut pas être résolu par Git. Et si les *'unresolved conflict'* ne sont pas sur la même ligne ou sur des lignes adjacentes, alors ils forment un 'automatically resolved conflict' ce qui signifie qu'ils sont résolus automatiquement par Git. Des *'unresolved conflicts'* se produisent également si un fichier est supprimé ou renommé pendant qu'il est modifié par un autre, s'il est renommé simultanément ou renommé en un nom de fichier existant et si deux utilisateurs ajoutent simultanément deux fichiers avec le même nom. Les utilisateurs doivent résoudre manuellement les *'unresolved conflicts'*.

En plus des outils l'EC 'asynchrone', plusieurs outils l'EC 'synchrone' ont été développés au début des années 1990 tels que GROVE - *'a textual multi-user outlining tool'* [7], ShrEdit - *'a multi-user text editor'* [21]. Cependant, les utilisateurs n'avaient pas d'expérience sur l'utilisation de ces premiers outils d'EC 'synchrones' qui étaient principalement utilisés dans les études de recherche d'EC. À cette époque, les utilisateurs préféraient toujours utiliser des outils d'édition 'traditionnels' avec lesquels ils étaient familiers pour [5] ou des outils l'EC avec des espaces de travail privés où ils pouvaient finaliser et revoir leur travail avant de les intégrer au document/projet

partagé. Les éditent de texte modernes tels que Google Docs [22], ShareLaTeX [23], Etherpad [24] sont populaires avec de nombreuses fonctionnalités utiles pour prendre en charge l'édition collaborative comme l'ajout de commentaires, la communication en ligne (chat), la révision des historiques et l'édition des journaux. Les utilisateurs sont plus habitués à travailler ensemble à l'aide de ces éditeurs collaboratifs en temps réel.

L'édition collaborative basée sur des éditeurs collaboratifs en temps réel est différente de celle basée sur le système de contrôle de version. Avec le système de contrôle de version, les utilisateurs doivent intégrer les modifications manuellement, c'est-à-dire que les utilisateurs décident quand et quels changements doivent être *'merged'*. En revanche, avec les éditeurs collaboratifs en temps réel, les modifications des utilisateurs sont automatiquement *'merged'* et le résultat est immédiatement visible par tous les membres du groupe. Les éditeurs collaboratifs en temps réel dépendent de l'approche *'Operation transformation'* [25] ou *'Conflict-free replicated data' (CRDT)* [26, 27] qui peut intégrer les changements simultanés de plusieurs utilisateurs. Cependant, des conflits peuvent survenir lorsque deux auteurs éditent de manière très rapprochée dans les dimensions temporelles et spatiales. Par exemple, deux auteurs tentent de corriger un mot en même temps, leurs corrections peuvent être dupliquées.

Le processus d'édition collaborative basé sur des éditeurs collaboratifs en temps réel peut être divisé en plusieurs *'session'*, y compris *'single-authored session'* et *'co-authored session'*. Ce processus de fragmentation nécessite un 'time interval' ou un *'maximum time gap'* prédéfini. Avec l'approche utilisant 'time interval', les activités d'édition sont divisées en plusieurs sessions de même durée, par exemple 15 minutes [28]. Avec l'approche utilisant *'maximum time gap'*, une séquence d'activités d'édition consécutives dans laquelle la 'distance temporelle' entre deux vérifications adjacentes est plus courte que le *'maximum time gap'* prédéfini est regroupée dans une session [29]. L'approche *'interval'* a un cas de pointe dans lequel deux modifications adjacentes qui sont proches dans le temps peuvent être divisées en différentes sessions. L'approche du *'maximum time gap'* surmonte ce cas de pointe.

Questions de recherche

D'après la littérature, seules quelques études ont analysé les changements parallèles et les conflits pour des projets basés sur des DVCS tels que Git [15, 16]. Ces études n'ont pas analysé les types de conflits lors des fusions et la fréquence des conflits à un niveau fin comme les lignes de conflit dans un fichier. Contrairement aux CVCS qui ont une journalisation centralisée qui suit toutes les activités d'édition des utilisateurs au niveau du fichier [14], Git ne fournit que les historiques de commit, y compris les commits et les fusions. C'est la vue d'ensemble des activités des utilisateurs au niveau d'un 'projet'. Plusieurs fichiers peuvent être mis à jour dans un commit et plusieurs fichiers peuvent être en conflit dans une fusion. Pour identifier les changements parallèles et les conflits au niveau du fichier dans Git, nous devons réintégrer tous les changements du développeur au cours du cycle de vie de développement du projet. En outre, les études précédentes sur les conflits en l'EC basées sur le système de contrôle de version n'ont pas étudié quels sont les mécanismes de résolution de conflit adoptés par les utilisateurs, tels que la restauration ou l'application de modifications à partir d'un ou des deux sites. Nous souhaitons étudier la fréquence à laquelle les utilisateurs utilisent la restauration comme résolution rapide. Nous voulons particulièrement analyser le conflit des lignes adjacentes.

La première partie de cette thèse vise à répondre aux questions de recherche ci-dessous :

Selon notre littérature, seules quelques recherches étudient l'édition collaborative basée sur l'analyse des journaux d'activités d'édition collaborative. Et la plupart d'entre elles sont toutes

1. À quelle fréquence chaque type de conflit textuel apparaît-il au cours des différentes phases de développement de projets basés sur Git ?
2. Comment les utilisateurs résolvent-ils ce type de conflit dans la pratique ?
3. En particulier, comment les utilisateurs résolvent-ils les *'adjacent-line conflict'* (conflits de lignes adjacentes) ?

des études 'contrôlées' dans lesquelles les utilisateurs doivent suivre certaines étapes spécifiques pour éditer en collaboration. Birnholtz et al [30] ont présenté une étude expérimentale de la maintenance de groupe dans l'édition collaborative à l'aide de Google Docs. Il s'agit de la première recherche selon la littérature qui envisage d'analyser les journaux d'édition. Sun et al [28] ont présenté une analyse détaillée des journaux d'activités sur deux ans de tous les employés de Google utilisant la suite Google Docs. Olson et al [29] ont examiné les traces du comportement d'écriture collaborative d'étudiants de premier cycle avancés dans un cours de projet utilisant Google Docs pour étudier *'how people write together now'*. D'Angelo et al [31] ont analysé les histoires d'une grande collection de documents édités dans Etherpad [24] pour étudier comment les gens écrivent sur nature. Dans ces études de recherche, 'l'intervalle de temps' ou 'l'intervalle de temps maximal' n'était pas encore bien défini. L'intervalle de 15 minutes et l'intervalle de temps maximum de 7 minutes ont été utilisés. Nous voulons examiner différents 'intervalles de temps maximum' pour voir lequel est le meilleur et comment choisir celui qui convient. En outre, une analyse détaillée des activités d'édition à l'intérieur des sessions, en particulier les *'co-author sessions'* qui contiennent probablement *'collaborative edits'*, peut nous apporter un aperçu précieux des sessions d'édition collaboratives. Et nous nous sommes également intéressés aux cas que deux auteurs éditent étroitement ensemble dans les dimensions temporelles et spatiales. En utilisant de petites *'time-position windows'*, nous pouvons examiner de plus près comment les gens gèrent ces cas de *'potential conflict'*.

Dans la deuxième partie de cette thèse, nous envisageons de répondre aux questions de recherche ci-dessous :

1. Comment choisir un *'maximum time gap'* approprié pour diviser les activités d'édition en sessions ?
2. Quelle est la caractérisation de la position dans le temps des sessions d'édition, spécialement les *'co-authored sessions'* ?
3. Dans les *'co-authored sessions'*, à quelle fréquence les *'potential conflicts'* se produisent-ils dans une extension (condition) de position temporelle ?

Contributions

La première partie : Conflits et résolutions dans les projets open source basés sur Git

Il existe d'autres études axées sur les changements parallèles et les conflits sur le projet de logiciel open source basé sur Git [15, 16], mais elles n'ont pas analysé les conflits à granularité fine au niveau des fichiers. En fait, ils ont considéré à la fois les conflits de fusion textuels et

les conflits d'ordre supérieur qui se sont produits lors de la compilation ou des tests comme des conflits textuels. Nous n'examinons pas le conflit sémantique (c'est-à-dire indirect ou d'ordre supérieur). Au lieu de cela, nous nous concentrons sur le conflit textuel et leurs mécanismes de résolution. Nous avons étudié les traces de quatre grands projets logiciels open-source : Rails [32], Iki Wiki [33], Samba [34] et Linux Kernel [35] afin d'analyser différents types de conflits textuel qui surviennent au cours des différentes phases de développement des projets et comment les développeurs résolvent ce type de conflit dans la pratique.

Nous avons constaté que parmi les différents types de conflits (*Content conflict, Remove/Update conflict and Naming conflict*), le *Content conflict* est le plus populaire. Il couvre de 46% à 90% du nombre total de conflits. Et en général, le *Integration rate* qui présente la proportion de mises à jour simultanées sur le total des mises à jour et le *Conflict rate* qui présente la proportion de mises à jour simultanées ayant entraîné un conflit non résolu sur le nombre de mises à jour simultanées dépendent de la façon dont le processus de développement (c'est-à-dire la collaboration) est géré. Par exemple, dans le projet Samba, le *Integration rate* est très faible (0,68%) mais le *Conflict rate* est très élevé (87,84%). Alors que dans le projet Linux-Kernel qui a le plus haut *Integration rate* (10,99%), le *Conflict rate* est le plus bas (4,86%). Ce résultat peut être expliqué par le fait que le projet Linux-Kernel tire de nombreux avantages des mainteneurs du sous-système et du modèle basé sur l'extraction tandis que le projet Samba utilise un référentiel partagé. De plus, nous avons effectué une analyse de l'intégration et du conflit à des périodes spécifiques qui sont quatre semaines avant et quatre semaines après la date de release. L'objectif est de mieux comprendre la collaboration pendant ces périodes actives. Nous avons constaté que le *Content conflict* est toujours le type de conflit le plus populaire (76% - 100%). Et la corrélation entre *' l'integration rate' et le 'conflict rate'* est nulle ou très faible (en utilisant le 'Spearman's rank correlation coefficient' avec les points de données collectés en fonction de la date de publication).

En ce qui concerne la façon dont les conflits sont résolus, nous avons mesuré la fréquence à laquelle les utilisateurs utilisent la 'restauration' comme solution rapide pour résolve les conflits. Le résultat montre la forte proportion d'activités de 'roll-back' dans les projets Samba (20,45%) et Rails (33,46%). Cependant, c'est très rare dans les projets Linux-Kernel (5.18%) et IkiWiki (5.21%).

Un type particulier de conflit textuel non résolu que nous avons analysé dans notre étude est le *'conflicting changes'* faisant référence à deux lignes adjacentes (appelées 'adjacent-line conflict'). Alors que d'autres systèmes de contrôle de version tels que Darcs [36], Subversion [19] résolvent ce type de conflit automatiquement, Git en revanche, signale un "conflit non résolu". Sur la base des résultats, les utilisateurs appliquent principalement toutes les modifications lors de la résolution des conflits de lignes adjacentes. Nous avons proposé que Git fusionne automatiquement ce cas et lance un message d'avertissement au lieu de signaler un conflit non résolu. En outre, nous voulions également étudier la fréquence à laquelle les utilisateurs reviennent à la version précédente (c'est-à-dire la version stable avant la fusion), comme mesure de la satisfaction des utilisateurs du résultat de la fusion.

La deuxième partie : Caractérisation de la position dans le temps de l'EC à l'aide de ShareLaTeX

Nous avons analysé les journaux d'édition collaboratifs collectés à partir d'un serveur ShareLaTeX [23] utilisé dans une école d'ingénieurs et anonymisés à des fins de confidentialité. Les journaux ont été réalisés à partir de travaux collaboratifs de groupes de trois ou quatre étudiants auxquels une tâche d'écriture collaborative a été attribuée. À partir des journaux, nous avons récupéré 892 documents dans lesquels seuls 108 documents ont plus d'un auteur. En examinant

différents *'maximum time gap'* de 30 secondes à 15 minutes, nous avons constaté que la distance de temps entre les sessions (c'est-à-dire *'l'external-distance'*) a une plage très large (jusqu'à 87 heures dans notre étude de cas) et en évaluant la distribution *'d'external-distance'* d'un très petit intervalle de temps, nous pouvons déterminer un *'maximum time gap'* approprié pour diviser les activités d'édition en sessions.

En ce qui concerne les activités d'édition dans les sessions d'édition, nous avons constaté qu'en général les sessions co-rédigées ont une durée plus longue et plus d'activités d'édition que les sessions à auteur unique. En outre, la distance temporelle entre deux modifications dans les 'co-authored session' est plus courte que dans les sessions avec un seul acteur. D'une autre manière, les utilisateurs sont plus productifs dans les sessions 'co-authored' que dans les sessions à 'single-authored'.

En nous concentrant sur les sessions co-rédigées, nous avons utilisé une 'time-position window' [30 secondes, 10 caractères] pour examiner les cas dans lesquels deux auteurs éditent en étroite collaboration. Plus en détail, nous avons examiné deux cas susceptibles d'entraîner un conflit : *'border cases'* et *'insertion case'*. *'Border cases'* fait référence aux cas dans lesquels deux auteurs différents éditent dans la bordure de deux zones d'édition proches qui leur appartiennent. Et *'Insertion case'* fait référence aux cas dans lesquels un auteur effectue des modifications entre deux modifications continues d'un autre auteur. Les résultats montrent que ces deux cas se produisent rarement : jusqu'à 5,04 % pour *'insertion case'* et jusqu'à 9,66 % pour *'border cases'*. Cela signifie que les gens modifient rarement sur un étroite place temporelle et spatiale. Cependant, ils sont plus susceptibles de devenir des conflits. Entre de 77,53 % à 91,51 % des *'border cases'* concluent à un 'conflict' et de 88,96 % à 100 % pour *'insertion case'*. À partir des résultats ci-dessus, nous suggérons que les outils d'édition collaboratifs (ShareLaTeX dans ce cas) devraient envisager d'avoir un mécanisme de sensibilisation pour ces deux types de *'potential conflicts'*.

Chapter 1

Introduction

In this chapter, we first introduce the Study Context of the thesis which is Collaborative Editing using different work modes. We then describe the research challenges of this study context from which we suggest our research questions. And finally, we present our contributions and the structure of the thesis.

1.1 Study Context

The term 'Computer-Supported Cooperative Work' (CSCW) was first introduced in 1984 by Irene Greif and Paul M. Cashman. Twenty people from different fields who were interested in how people work explores technology's role in the work environment and used the term CSCW to describe it [37]. CSCW can be divided into two main areas, associated with *'Computer-Supported'*(CS) and *'Cooperative Work'*(CW), respectively. Together with 'CSCW', the term 'Groupware' or 'Collaborative software' was described as *'intentional group processes plus software to support them'* by Peter Johnson-Lenz and Trudy Johnson-Lenz [38]. The authors then restricted their definition to the 'computer-based system'. Meanwhile, Carstensen and Schmidt [39] consider Groupware as part of CSCW which is associated with the 'CS' part of CSCW. They claim that CSCW addresses *'how collaborative activities and their coordination can be supported by means of computer systems'*. Similarly, according to Ellis et al [7], CSCW looks at how groups work and seeks to discover how technology (specially computer technologies) can help them work. They then defined Groupware as *'computer-based systems that support group of people engaged in a common task (or goal) and that provide an interface to shared environment'*. The authors also presented three basic concepts that CSCW research and Groupware design should focus on : *'Communication', 'Collaboration' and 'Coordination'*. *'Communication'* concerns about how Groupware systems support people in communication during common tasks. It can be divided into two types of supporting : asynchronous communication (different time) and synchronous communication (real-time). *'Collaboration'* refers to the shared environments (i.e the group's context) that offer up-to-date information about all collaborator's activities. *'Coordination'* refers to the integration of group's activities. These concepts are widely used in follow-on researches.

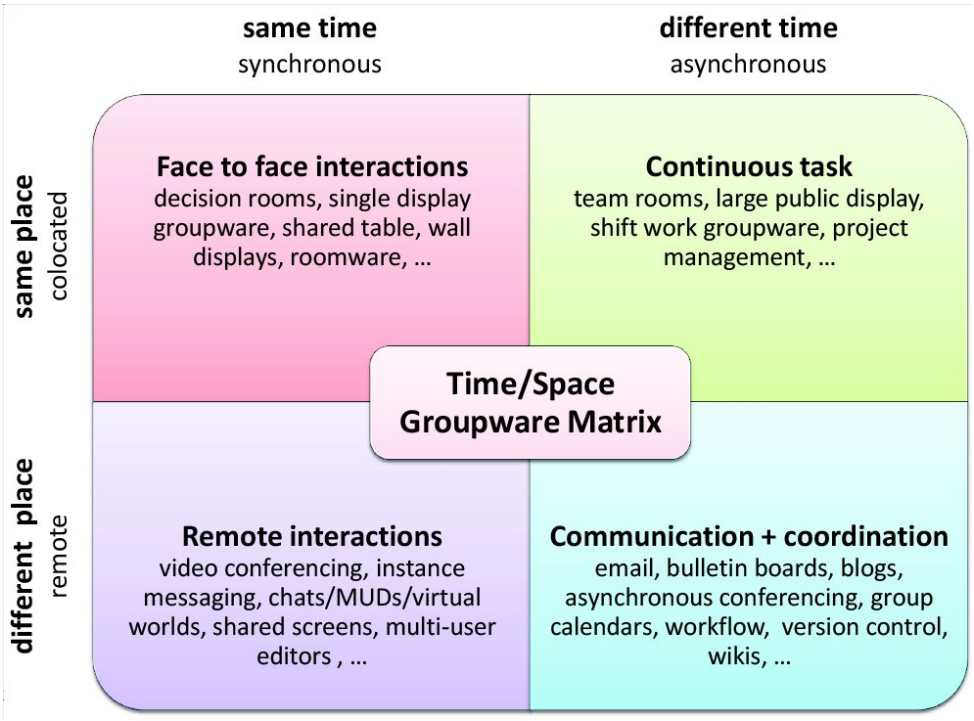
Different cooperative works were considered by CSCW researchers. According to the *'Application-Level taxonomy* [7], Groupware systems could be classified into different categories including : Message Systems, Multi-user Editor, Group Decision Support Systems and Electronic Meeting Rooms, Computer Conferencing, Intelligent Agents, Coordination Systems. Although some of these categories overlap with each other, the taxonomy has given a general view of different

cooperative works that CSCW and there by Groupware considered to support. Among these cooperative works, 'Collaborative Editing' (also known as 'Collaborative Writing') is widely practised in both academia and industry as it brings many benefits [4].

Editing itself is a complex and time-consuming task. Editing collaboratively makes it more complex with many additional activities such as coordinating, communicating, negotiating, monitoring [4]. The authors need to coordinate between multiple viewpoints and work efforts [3]. They also need to deal with social relationships when pointing out collaboration problematic [40]. Despite its complexities, Collaborative editing is used more and more commonly because of its worthy benefits. Different terms were used to describe Collaborative Editing (CE) such as : *group writing* [1], *collaborative authoring* [8], *collaborative writing* [9, 10, 4, 41], *writing together* [3, 29] and itself collaborative editing [6, 29, 31]. However, they all commonly imply to *'the process of two or more people working together to create a complex document'* [41]. Traditional collaborative editing, for example, group members edit on their personal word-processor such as Microsoft Word and share over email as attached file. If they all follow a clear working plan which assigns who to write what and when, the integration is not difficult in this case as there is only a single person at a time for doing changes (i.e edit the document) [5]. However, if they write in isolation from each others, the integration process becomes complex as two or more people may edit in parallel the same part of the document. Many authors with supporting from editor tools try to improve their performance, reduce production time and also improve the final result by editing collaboratively. For example, a group of developers use version control system such as Git to work in collaboration in a software project. Version control systems allow each member of a group to work in isolation (i.e without disbursing from others) and integrate their works later. The integration (i.e the merge) is handled automatically by version control system. However, in some case, it results in conflict and needs user inventions to resolve.

Collaborative editing has long captured the attention of Computer-supported cooperative work(CSCW) researchers. Many researches have been conducted, many tools had been built to support and improve collaborative editing. Most of researches in the 1990s and the early 2000 focus on describing characteristics of CE and its advantages/disadvantages [1, 3, 4, 5] ; classifying work-modes [6, 4] ; building CE taxonomy [7, 8, 3, 4, 5] and proposing requirements for tools supporting CE [3, 9, 10]. They were all based on interviewing people who had participated in some collaborative editing projects. A notable work from these researches was presented by Posner et al[3] in which the authors present a taxonomy of CE in terms of four components : Role, Activity, Document Control Method and Writing strategy. Each of them provides a different perspective of CE. *Roles* describe the part played by each individuals on the editing group such as : writer, consultant, editor and reviewer. *Activities* describe the actions performed while people work on the project, including : brainstorming, researching, planing, writing, editing, reviewing. *Document Control Methods* describe how the writing process is managed and coordinated. It includes : 'centralized method' in which one individual controls the document throughout the project, 'Relay method' in which one person at a time controls the document and the control is passed between all team members,'Independent method' in which each member works on different parts of the document and 'Shared method' in which all team members can access and edit simultaneously. While *Document Control Methods* focus on the *object*(the document), *Writing Strategies* focus on the *process* of creating document. They are closely related. *Writing Strategies* can be 'Single writer' or 'Scribe' in which one member writes the document based on discussions of all members in the group ; 'Separate writers' in which the documents is broken up into parts and each member is responsible for a different part ; 'Joint writing' in which all members write together. And finally, it can be 'consulted strategy' which is not a complete strategy itself and often used in combination with other strategies. The authors then based on the taxonomy

and the interview, suggested a sets of 13 requirements for tools/systems supporting CE which includes supporting synchronous and asynchronous editing. This taxonomy was extended by Lowry et al [4] with some refinements and addition of Work Mode component which describes the physical distance between members of the group (i.e. same location or different location) and the synchronicity of editing activities (i.e synchronous or asynchronous editing). The *Work Mode* component was also presented previously by Ellis et al [7] as the *'CSCW Matrix'* (or the *'Time Space taxonomy'*) including four categories : [Same location, Same time], [Same location, Different time], [Different location, Same time], [Different location, Different time]. Figure 1.1 presents the *'CSCW Matrix'* and with annotations for each category.



Source: Wikipedia - Computer-supported cooperative work[42]

FIGURE 1.1 – The CSCW Matrix

The *'CSCW Matrix'*, however, supposes that all group members use the same work mode (asynchronous or synchronous) for the whole editing time. This is not how actually people work together. Minor et al[6] showed that people use different writing strategies when editing a document or program together. For instance, two or more users edit a paper together, they do not edit simultaneously all the time. One user may stop editing for some time (a day for example) and come back later for editing. Similarly, Posner et al [3] confirmed that both synchronous and asynchronous strategies are used in different phases of CE project and suggested that CE systems must support a smooth transition between two styles. Further than the standard 'asynchronous' and 'synchronous' work modes which consider collaborative work as a single stream of activity, Dourish et al[11] presented the 'multi-synchronous' work mode which considers collaborative works as 'multiple, parallel streams of activity'. In 'multi-synchronous' work mode, participants work in isolation while working activities proceed in parallel (divergence). Their individual work will be integrated (synchronization) periodically. If the time interval between synchronizations is small enough in comparison to editing actions of the users, all collaborators can see editing activities of others after a very short time. In this case, 'multi-synchronous' work mode is very close

to the standard 'synchronous' work mode. And in the opposite case, if the synchronization takes place less frequently (after hours, days or weeks), 'multi-synchronous' is similar to the traditional 'asynchronous' work mode. In another way, we can consider 'asynchronous' and 'synchronous' as two aspects of 'multi-synchronous' with different period of synchronization.

Conflict may happen in integration phase as changes made by users during divergence phase can overlap with each other. The longer period of divergence is, more conflicts are likely to happen. Conflicts might occur at textual or semantic level. Textual conflict is the result of textual merging concurrent changes of two different individuals which are, depending on the context, at same line or two adjacent lines of a shared file. Textual conflict can be detected by most of systems supporting CE. Semantic conflict (also known as 'Higher-order' or 'In-direct' conflict) is the conflict at semantic level. For instance, in case of software development based on Git, concurrent changes of two developers are merged successfully (i.e without textual conflict). However, if the software can not be compiled or gets test failures, the integration of these concurrent changes is considered as semantic conflict.

Conflicts are costly as they delay the development process. It calls to the needs of a better understandings on how often and when conflicts are more likely to happen when people working together, especially in 'asynchronous' work mode. Several studies had been conducted, focused on parallel changes and conflicts in different systems supporting collaboration such as [12] (Ficus file system), [13] (Lucent Technologies' 5ESS), [14] (CVS), [15, 16] (Git). All these systems support tracking integration (merging) changes, it's a valuable corpus to analyse besides the interviews based data. Among them, Git [17] - a decentralized version control system became popular in software development community around 2005. While centralized version control systems such as CVS [18] or Subversion [19] rely on a client-server architecture, Git relies on the peer-to-peer architecture. In this architecture, each client keeps a complete copy (clone) of the shared project. Users can work in isolation on their local repository and can synchronize their works with ones of other collaborators. A 'merge' is performed when users choose to synchronize. Git, similar to others version control system, uses a textual merging technique [20] in which 'lines' are considered as indivisible units. When a file is edited in parallel, we call changes that made by different collaborators are *'conflicting changes'*. If the *'conflicting changes'* refer to the same line or adjacent lines of the file, then they form a 'unresolved conflict' which can not resolve by Git. And if the *'conflicting changes'* are not in the same line or adjacent lines, then they form an 'automatically resolved conflict' which means they are resolved automatically by Git. 'Unresolved conflicts' also happen if a file is deleted or renamed while being modified by other, if it is renamed concurrently or renamed to an existing file-name and if two user concurrently add two files with the same name. Users need to resolve 'unresolved conflict' manually.

Beside 'asynchronous' CE tools, several 'synchronous' CE tools were developed in the early 1990s such as GROVE - *a textual multi-user outlining tool*[7], ShrEdit - *a multi-user text editor*[21]. However, users did not have experience on using these early 'synchronous' CE tools which were used mostly in CE research studies. At this time, users still preferred using 'traditional' editing tools which they were familiar to [5] or CE tools with private work-spaces where they can finalize and review their work before integrating them to the shared document/project. Modern word processors such as Google Docs [22], ShareLaTeX [23], Etherpad [24] are popular with many useful features to support collaborative editing such as adding comments, in-line communication (chat), revision histories and editing logs. Users are more familiar with working together using these real-time collaborative editors.

Collaborative editing based on real-time collaborative editors is different than CE based on Version control system. With version control system, users need to integrate changes manually, ie. users decide when and what changes should be 'merged'. In contrast, with real-time collaborative

editors, changes from users are 'merged' automatically and the result is visible immediately to all group members. Real-time collaborative editors depend on the Operation transformation [25] or Conflict-free replicated data (CRDT) [26, 27] approach which can integrate concurrent changes from multiple users. However, conflicts may potentially happen when two authors edit very close together in both time and position dimensions. For instance, two authors try to correct a word at the same time, their corrections can be duplicate.

The process of collaborative editing based on real-time collaborative editors can be split into several *'session'*, including *'single-authored session'* and *'co-authored session'*. This fragmentation process requires a predefined 'time interval' or 'maximum time gap'. With the approach using 'time interval', editing activities are split into several sessions with the same length, for instance, 15 minutes [28]. With the approach using 'maximum time gap', a sequence of consecutive editing activities in which the 'time-distance' between two adjacent edits is shorter than the predefined 'maximum time gap' are grouped into a session [29]. The 'interval' approach has an edge case in which two adjacent edits which are close in time can be split into different sessions. The 'maximum time gap' approach overcomes this edge case.

1.2 Research Questions

From the literature, only few studies analysed parallel changes and conflicts for projects based on DVCSs such as Git [15, 16]. These studies did not analyse the types of conflicts during merges and the frequency of conflict at fine-grained level such as conflict lines in a file. In contrast to CVCSs which have a centralized logging tracking all user editing activities at file level [14], Git provides only the commit histories including commits and merges. It's the overview of user activities at 'project' level. Several files can be updated in a commit and several files can be in conflict in a merge. To identify parallel changes and conflicts at file level in Git, we need to re-integrate all developer's changes during the development life cycle of the project. In addition, previous studies about conflict in CE based on version control system did not study what are the conflict resolution mechanisms adopted by users such as roll-back or applying changes from one or both sites. We are interested to study how often users use roll-back as a quick resolution. We particularly want to analyse the adjacent-lines conflict.

The first part of this thesis is aimed to answer the questions presented in Figure 1.2.

1. How often each type of textual conflict appears during different development phases of Git based projects ?
2. How do users resolve these type of conflict in practice ?
3. In particular, how do users resolve 'adjacent-line conflict' ?

FIGURE 1.2 – Research questions, Chapter 3

According to our literature, only few researches study about collaborative editing based on analysing the logs of collaborative editing activities. And most of them are all 'controlled' studies in which users have to follow some specific steps to edit collaboratively. Birnholtz et al [30] presented an experimental study of group maintenance in collaborative editing using Google Docs. This is the first research according to the literature that consider to analyse editing logs. Sun et al [28] presented a detailed analysis of activities logs over two years of all Google employees using Google Docs suite. Olson et al [29] examined the traces of collaborative writing behavior of advanced undergraduates in a project course using Google Docs to investigate on *'how people*

write together now'. D'Angelo et al [31] analysed the histories of a large collection of documents edited in Ethepad[24] to study how people writing in the wild. In these research studies, the 'time interval' or the 'maximum time gap' was not yet well defined. The '15 minutes interval' and the '7 minutes maximum time gap' were used. We want to examine different 'maximum time gaps' to see which one is better and how to choose a suitable one. Besides, a detailed analysis of editing activities inside sessions, specially *'co-author sessions'* which probably contain *'collaborative edits'*, can bring us valuable insight of collaborative editing sessions. And we were also interested in the cases that two authors edit closely together in both time and position dimensions. Using a small 'time-position windows', we can have a closer look at how people manage these 'potential conflict' cases.

Figure 1.3 presents our research questions that we consider to answer in the second part of this thesis.

1. How to choose a suitable 'maximum time gap' to split editing activities into sessions ?
2. What is the time-position characterization of editing sessions, specially 'co-authored sessions' ?
3. Inside 'co-authored sessions', how often 'potential conflicts' happen within some time-position extension (condition) ?

FIGURE 1.3 – Research questions, Chapter 4

1.3 Contributions

1.3.1 The first part : Conflicts and resolutions in Git-based open-source projects

There are other studies focused on parallel changes and conflict on Git-based open-source software project [15, 16], however, they did not analyse fine-grained conflicts at file level. In fact, they considered both textual merging-conflicts and higher-order conflicts which raised during compiling or testing as textual conflicts. We do not investigate the semantic (i.e indirect or higher-order) conflict. In stead, we focus on the textual conflict and their resolution mechanisms. We studied the traces of four large open-source software projects : Rails [32], Iki Wiki [33], Samba [34] and Linux Kernel [35] in order to analyse different types of textual conflict which arise during different development phases of the projects and how developers resolve this type of conflict in practice.

We found that among different types of conflicts (*Content conflict, Remove/Update conflict and Naming conflict*), *Content conflict* is the most popular. It covers from 46% to 90% of the total number of conflict. And in general, the *Integration rate* which presents the proportion of concurrent updates over the total updates and the *Conflict rate* which presents the proportion of concurrent updates resulted in unresolved conflict over the number of concurrent updates depend on how the development process (i.e the collaboration) is managed. For instance, in Samba project, the *Integration rate* is very low (0.68%) but the *conflict rate* is very high (87.84%). While in Linux-Kernel project which has the highest *Integration rate* (10.99%), the *conflict rate* is the lowest one (4.86%). This result can be explained by the fact that Linux-Kernel project gets many advantages from the subsystem maintainers and pull-based model while Samba project uses a shared repository. In addition, we conducted an analysis of the integration and conflict

at specific time periods which are four weeks before and four weeks after the release date. The objective is to gain a better understanding about collaboration during those active periods. We found that *Content conflict* is still the most popular type of conflict (76%- 100%). And the correlation between *Integration rate and Conflict rate* is none or very weak (Using Spearman's rank correlation coefficient with data points collected based on release date).

Regarding to how conflicts are resolved, we measured how often users use 'roll-back' as a quick resolution for conflicts. The result shows the high proportion of 'roll-back' activities in Samba (20.45%) and Rails (33.46%) projects. However, it's very rare in Linux-Kernel (5.18%) and IkiWiki (5.21%) projects.

One particular type of unresolved textual conflict that we analysed in our study is the *'conflicting changes'* refering to two adjacent lines (called 'adjacent lines conflicts'). While other version control systems such as Darcs [36], Subversion [19] resolve this type of conflict automatically, Git in contrast, signals an 'unresolved conflict'. Based on the results, users mostly apply all changes when resolving adjacent lines conflicts. We proposed that Git should merge this case automatically and throw a warning message instead of signaling an unresolved conflict. In addition, we were also interested in studying how often users roll back to previous version (i.e the stable version before the merge), as the measurement of how user satisfaction of the merge result.

1.3.2 The second part : Time-position characterization of CE using ShareLaTeX

We analysed collaborative editing logs which were collected from a ShareLaTeX [23] server used inside an Engineering school and anonymized for privacy purpose. The logs were conducted from collaborative works of groups of three or four students which were assigned a collaborative writing task. From the logs, we retrieved 892 documents in which only 108 documents have more than one authors. By examining different 'maximum time gaps' from 30 seconds to 15 minutes we found that the time distance between sessions (i.e *'external-distance'*) has a very wide range (up to 87 hours in our case study) and by evaluating the distribution of *'external-distance'* of a very small time gap, we can determine a suitable 'maximum time gap' to split editing activities into sessions.

Regarding editing activities inside editing sessions, we found that in general co-authored sessions have longer length and more editing activities than single-authored sessions. Besides, the time-distance between two edits in co-authored sessions is shorter than in single-authored sessions. In another way, users are more productive in co-authored sessions than in single-authored session.

Focusing on co-authored sessions, we used a [30 seconds, 10 characters] time-position window to examine the cases in which two authors edit closely together. In more details, we examined two cases which potentially result in conflict : *'border cases'* and *'insertion case'*. *'Border cases'* refers the cases in which two different authors edit in the border of two close editing areas that belong to them. And *'insertion cases'* refers to the cases in which one author does some edits between two continuous edits of another author. The results show that these two cases happen rarely : up to 5.04% for *'insertion cases'* and up to 9.66% for *'border cases'*. It means that people rarely edit closely in both time-position. However, they are more likely to become conflicts. It's from 77.53% to 91.51% of *'border cases'* result in 'conflict' and it's from 88.96% to 100% for *'insertion cases'*. From above results, we suggest that collaborative editing tools (ShareLaTeX in this case) should consider to have an awareness mechanism for these two types of 'potential conflict'.

1.4 Structure of the Thesis

The rest of this thesis is organized as follows. Chapter 2 provides the basic knowledge about Collaborative Editing (CE), Conflicts in CE and the reviews of previous works which are related to the problems considered in this thesis. In Chapter 3 we deal with Merge Conflicts and Resolutions in Git based projects. Chapter 4 presents our study about the Time-position characterizations of Conflicts in collaborative editing using ShareLaTeX inside an Engineering school. And finally, Chapter 5 concludes with a summary of this thesis, discusses its achievements as well as its limitations and suggests some possible future research directions.

Chapter 2

State of the art

This chapter is organized as follow : In 'Basic notions' section (section 2.1), we present some basic knowledge of Collaborative Editing including 'version control system' and 'real-time web-based collaborative editor'. We then review the previous studies on Collaborative editing based on Version control system and Real-time web-based collaborative editor. (sections 2.2 and 2.3). We also present about Conflict awareness in CE (section 2.4). Finally in 'Chapter conclusion' section (section 2.5), we stated our research objectives.

2.1 Basic notions

2.1.1 Version control system

Version control, also known as 'Revision control' or 'Source control', is the management of changes to files. It is very popular not only in software development but also in many collaborative works. According to Minor and Magnusson [6], '*version control* is a crucial technique for all collaborative design environments and editors'.

Version control uses a *repository* to store all changes made to a file or a set of files over time and allows users to make edits on a *working copy* (also known as a *checkout*). A *working copy* is a personal copy of a *repository* where its owner can make edits without transmitting them immediately to the other members of the group. These changes might be transmitted at a later time. Figure 2.1 shows a simple example of how *version control* works. When users finish and are happy with their editing, they can *commit* their changes to a *repository*. *Commits* from users are stored as *versions* in the *repository* which are usually associated with a time-stamp and labeled appropriately. A *version* is not changeable once it has been created. New changes made on any existing *version* will be submitted to a new *version*. It's possible that a *working copy* is outdated, i.e missing some changes from another *working copy* which are committed to the *repository* recently. In this case, users can *update* their *working copy* to the latest *version* on the *repository*.

Version control provides access to the historical *versions* of a project. Figure 2.2 illustrates the simplest case in which changes are made and committed in linear order : 'Version 1'(C1) is created from changes made on 'Initial Version'(C0), 'Version 2'(C2) is created from changes made on 'Version 1'(C1) and so on. *Version control* allows multiple people to work simultaneously on a project. Each user edits on his/her own working copy in isolation with others. And when they want to *commit* their works, they need to *merge* edits from different *working copies*. In this case, the version history is split (called *'branching'*) and merged again. Figure 2.3 shows an example of branching and merging version history in which the 'Initial Version'(C0) is edited in parallel by

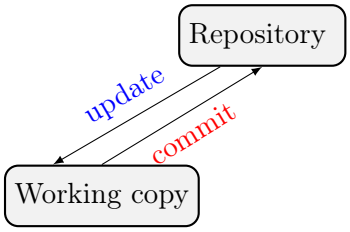


FIGURE 2.1 – Repository and Working copy

two users. 'Version 1' and 'Version 2' are committed as a result of this *'branching'*. At this point, users can keep the version history diverging or integrating (*merge*) these two versions. Users also can compare the differences between two versions using 'diff' algorithm. In this example, 'Version 1' and 'Version 2' are *merged* into 'Version 3'. The version history continues growing up with new changes committed as 'Version 4'. Users can *roll back* to previous *version*. In another way, if users make a mistake on the latest *commit* or the *merge*'s result is bad, they can recover to previous stable version. Normally, the *roll back* action creates a new *commit* that contains exactly the same contents of the source *commit*. For instance, 'Version 3' (i.e the merge result) contains some errors that are difficult to fix and user wants to *roll back* to 'Version 1'. 'Version 4' is the results of the *roll back* action, then it will be 'equal' to 'Version 1'. In version control system such as Git [17], 'Version 4' and 'Version 1' 'share' all the contents (i.e 'Version 4' does not stores a copy of all content objects of 'Version 1' but it stores a set of 'references' to them).

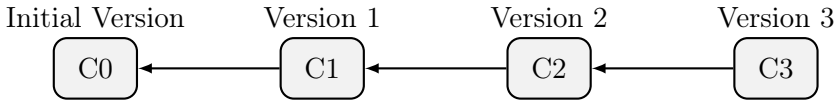


FIGURE 2.2 – Linear history

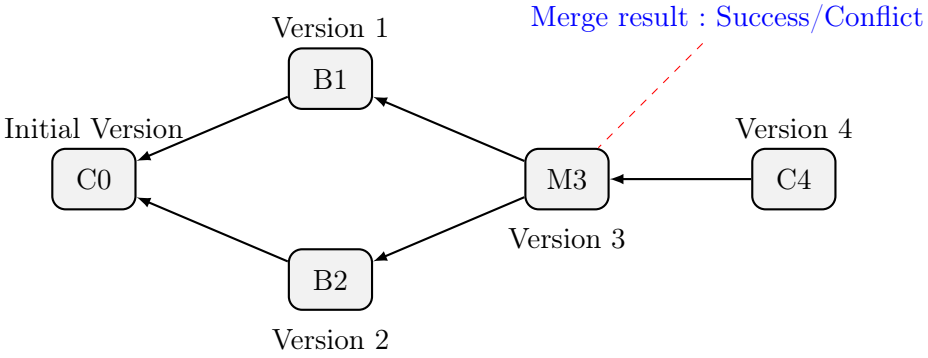


FIGURE 2.3 – Branching and Merge

Merge conflict and resolution

Version control systems usually are able to *merge* automatically parallel changes made by two different users. Coming back to the example in Figure 2.3, if changes made in 'Version 1' and 'Version 2' belong to different files or different parts of a file, the *merge* is handled by the version control system. However, if changes are made to the same parts of a file (i.e the same line

or adjacent lines), the system can not integrate changes automatically. This case is called *textual conflict* or *update-update conflict* and it needs user intervention to be resolved. In another way, when the automatic or semi-automatic merging operator results in conflict, users need to resolve the conflict manually. In Figure 2.3, if the merge between 'Version 1' and 'Version 2' results in conflict on a file, users then need to decide which 'version' of that file is better to be picked up or combine changes from both versions. They also can eliminate all changes from both versions and make new changes or even 'roll back' to the 'Initial Version'. Most of version control systems support 'roll back' actions as it is very useful when the conflict is complex and difficult to resolve manually.

Merge conflict also happens in some other cases. If two files are created with the same name, in the same folder (i.e they have the same full file name including full-path and file-name), we have *naming conflict*. If the file is deleted by one user while it's being edited by another one, we have *update-remove conflict*. *Naming conflict* (i.e *file naming*) is usually resolved by the system while *update-remove conflict* usually requires human intervention [12].

Centralized/Decentralized Version control system

Version control systems can be classified as *Centralized Version control system* (CVCS) or *Decentralized/Distributed Version control system* (DVCS). Figure 2.4 and Figure 2.5 give an overview of how they work. In *Centralized version control* (Figure 2.4), there is only one 'Central repository' which stores all versions of the project. Each user *checkouts* and edits files to his/her own *working copy*. And as soon as he/she *commits* his/her changes to the 'Central repository', it's possible for other members of the team to see his/her works. They then can *update* new changes to their *working copy*.

In contrast to *Centralized version control*, *Decentralized version control* has many repositories. Users have their own repository in local machine where they can *checkout (update)* a version from (for editing) and *commit* their edits to. Note that *commit* and *update* commands in a local repository do not affect other repositories (i.e changes made and committed locally). Each repository is possible to contain some versions that other repository are not updated with. Users can choose which repository to share their changes with or to update from. These processes are called *push* (share with) and *pull* (update from). Figure 2.5 presents clearly the difference between *update/commit* and *pull/push*.

This subsection 2.1.1 gives us the general understanding about 'version control system'. In the next subsection 2.1.2, we are going to describe more details about Git, a decentralized version control systems which is very popular in software development communities.

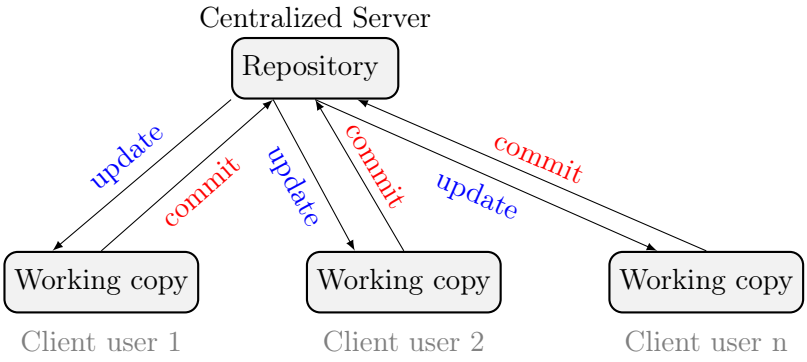


FIGURE 2.4 – Centralized Version control system

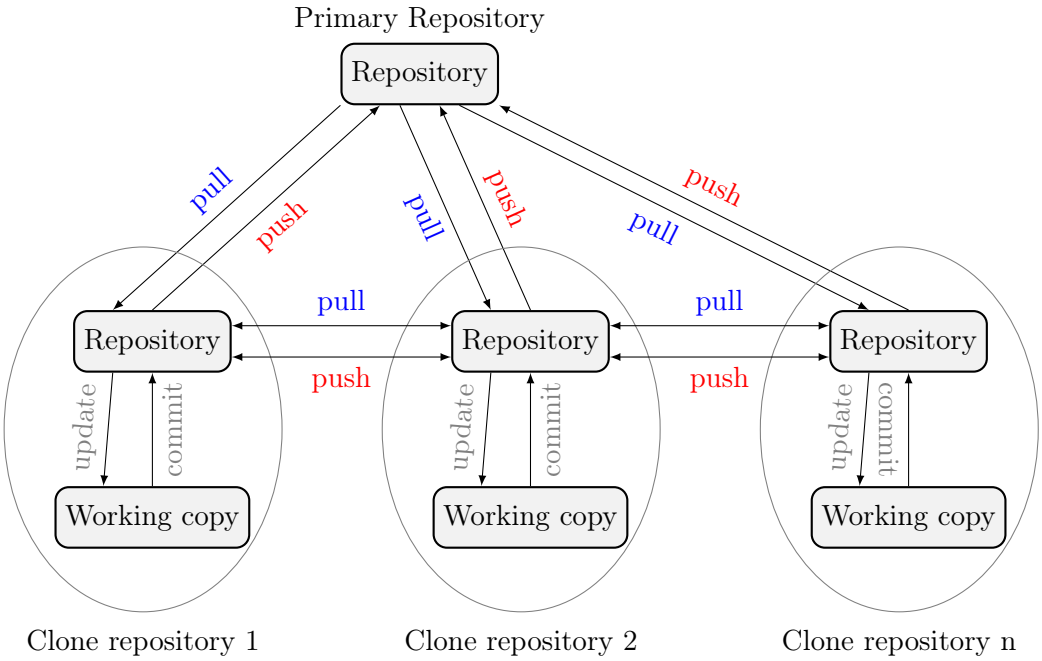


FIGURE 2.5 – Decentralized Version control system

2.1.2 Git- A decentralized version control system

Git [17], a decentralized (also called 'distributed') version control, was developed and maintained by Linus Torvalds, Junio Hamano and other kernel developers in April of 2005. It is distributed as an open-source software under the terms of GNU General Public License [43]. At the begining, Git was used only for Linux kernel project [35]. It then spread rapidly and quickly became the main version control of many other open source project such as Fedora and Samba [34]. Since 2008, it has begun to be spread outside the Linux world and became more and more popular in software development communities. Many medium and large size projects such as Ruby on Rails [32], IkiWiki [33] and many other open-source project started using Git. We will go through the important characteristics of Git and how it works to have a better understanding about how Git supports collaborative work.

Git Data Model

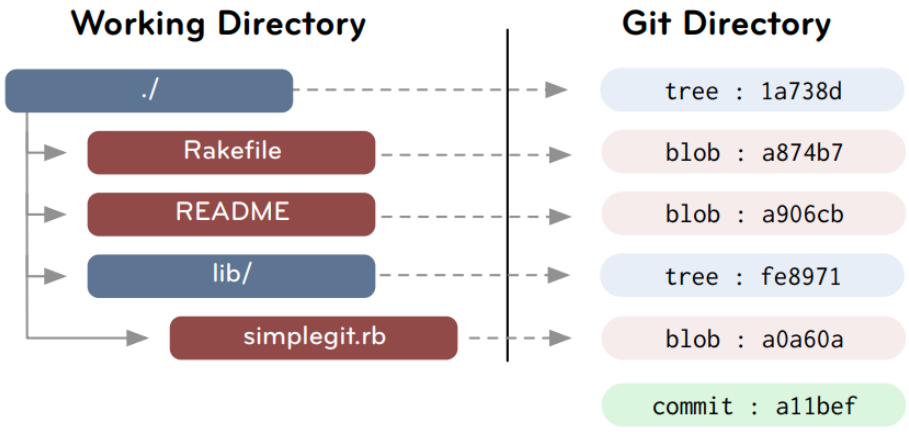
Git has four main types of objects : **blob, tree, commit** and **tag** in which the first three types are the most important. Below is the description of these object types.

— *Blob (binary large object)* : The contents of files are stored as *blobs*. A *blob* is identified by the 'SHA-1 hash' of its contents plus a small 'header'. *Blobs* don't have other metadata such as file-name or mode. It means that for two files with different file-names and even stored in different directories that have exactly the same content, Git will store only one *blob*.

— *Tree* : A *tree* is a simple list of *trees* and *blobs* including the names and the modes of those trees and blobs. In another way, a *tree* is similar to a directory which can contain files (*blobs*) or other directories (*trees*). The content section of a *tree* object is a text file that lists the *[mode, type, SHA-1, name]* of all contained objects.

— *Commit* : A *commit* is much like a *tree*. It points to a *tree* objects (which is the top-level source directory) along with *a timestamp, a message, an author, a committer and any parent commit that directly connected (preceded) to that commit.*
— *Tag* : A *tag* is an object that provides a permanent shorthand name for a particular *commit*. It contains *an object, type, tagger and a message*. Normally, the 'type' is the "commit" and the 'object' is the SHA-1 of the particular *commit*. In another way, a *tag* contains a reference to a *commit* object with some meta-data related to that *commit* object.

Each object is identified by a SHA-1 (Secure Hash Algorithm 1) hash of its contents plus a small 'header'. SHA-1 is a cryptography hash function which takes the input and *'produces 160 bits hash value - typically rendered as a hexadecimal number, 40 digits long'* [44]. Not only Git but also other VCS such as Mercurial use SHA-1 to identify versions and to ensure that the immutable objects are not changed when they are already stored. In Git, the SHA-1 hash value is computed and used as the object's name. New objects are stored using 'Zlib', a software library used for data compression. Objects can be combined into packs using delta compression (i.e store blobs by their changes related to other blogs) to save disk space. All Git objects are stored in the *'Git Directory'*. Figure 2.6 illustrates a mapping between a 'Working Directory' into a *'Git Directory'*. Files are presented as *blobs*, directories (sub-directories) are presented as *trees* and a *commit* presents the current *'version'* of the 'Working Directory'. If users edit files in this 'Working Directory' and submit their modification, a new *commit* (i.e a new *'version'*) is created and the current *commit* is referenced as its *'parent'*. Besides, Git stores each version of a file as a unique *blob*. It means that if a file in the 'Working Directory' is not changed over several *commits*, Git stores it once in a *blob* and that *blob* is 'shared' by those two *commits*. Note that in Figure 2.6, instead of using full 40 digits values,the SHA-1 hash values are presented in their short-form with their first 6 digits only.

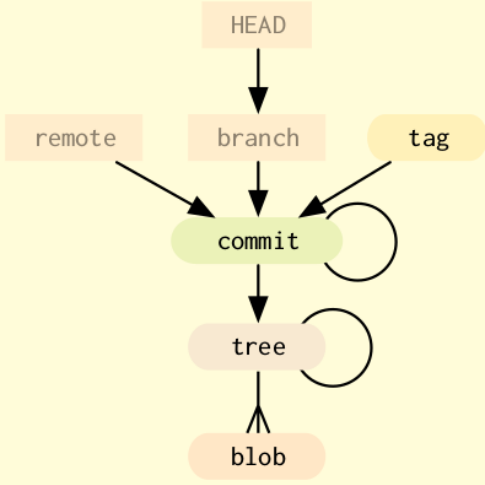


Source: Git Internals [45]

FIGURE 2.6 – Git Directory

In addition to four main object types which are 'immutable' (i.e they cannot be changed), Git also has the *'references'* object which is 'mutable'. The *'references'* are similar to the *tag* objects which point to a particular *commit* but they can be changed. Two popular *'references'* in Git are the 'branch' and the 'state of the remote'. They are stored in the '.git/refs/heads/' directory and simply contain the SHA-1 of the most recent commit (i.e the 'HEAD') of the branch that they

point to. Figure 2.7 presents the Data model of Git based on 4 immutable objects *'blob, tree, commit, tag'* and the mutable *references*. The link between *tree* and *blob* implies the 'containing'. A *tree* can also contain other *trees* ; this link is presented by the circle from the *tree* object to itself in Figure 2.7. A *commit* usually links to a top-level *tree* and if it is not the first *commit*, it also has a link to its 'parent' *commit*. The *tag* and the *branch* objects all point to a particular *commit*. However, when a child *commit* of the current *commit* is created, only the *branch* object is changed and points to the new *commit*. The *tag* object is unchanged and still points to the old *commit*. The *HEAD reference* object is a simple link to the current active *branch*. The *remote* (i.e 'state of the remote') object point to the current state of the branch in the remote repository. In another way, it usually points to the latest *commit* of the remote repository that the local repository is cloned from. It is changed only when user pulls changes from the remote repository.



Source: Git Internals [45]

FIGURE 2.7 – Git Data Model

Basic workflows in Git

In this subsection we go through some basic workflows in Git, and also explain the related Git operators (Git commands). We then introduce an intuitive example to illustrate these basic workflows.

- Getting a repository : There are two ways to get a Git repository in your local machine. In the first way, you can initialize a new Git repository by using *git init* command. This command will create a new repository at the current working directory or the directory that you provide when running the command. The second way is using *git clone* command to download an existing repository. You need to provide the address (i.e 'url') of the repository that you want to 'clone'.

— *git init* : initialize a local repository.
— *git clone* : download (clone) an existing remote repository. A remote reference object which points to the remote repository is created in this case.

- Edit and save (commit) to your local repository : After getting a Git repository to your local machine, you can start to edit files in your working directory and save your work to the local Git repository. To do this, we need to register all changed files with Git using 'git add' and then using 'git commit' to do the 'saving' action. The 'git add' command adds files into the 'index' stage which is the layer between the working directory and the repository. If a file is not

added, it only exists in the working directory but is not included in the *commit tree*. Git also provides the 'git commit -a' command which automatically adds and commits all changed files to a local repository. And if you make some mistakes in a commit and you want to 'delete' it, Git supports the *git reset* and *git revert* commands to do this. While *git reset* removes all commits on the way back to the specific commit, *git revert* keeps all these commits and creates a new commit. This commit is a 'copy' of the specific commit that users want to 'revert' to.

— *git add/rm* : puts/removes file(files) into the index stage
— *git commit* : commits staged changes to a local branch.
— *git reset* : makes the current branch point to some specific version. Users can choose to reset only the 'HEAD' or also the 'index stage' and the the 'current working directory'.
— *git revert* : reverts to a specific version by create a new commit which is a 'copy' of that specific version.

- Branching and Merging : Branching is one of the most effective features of Git. When getting a repository, you have only the 'master' branch with a 'branch' reference object and additionally a 'remote' reference object. Suppose that you started editing and had some commits to the 'master' branch. Then you get a new idea and want to try it without touching to the work you have done on the 'master' branch. With Git, you can simply create a new branch using *git branch* and start working with your new idea on it. You can create as many branches as you want. In Git, adding a new branch is simply adding a new 'branch' reference object in '.git/refs/heads/' directory of the current working branch. To switch among different branches, Git provides the *git checkout* command. When you finish developing your new ideas, you can 'merge' your work back to the 'master' branch. Git provides the *git merge* command to do this work. The merge process can result in either success or conflict. We will talk about the merge conflict later on.

— *git branch* : creates a new branch with provided name.
— *git checkout* : replaces the current working files with files from a branch.
— *git merge* : merges changes from a given branch into the current branch.

- Update your local repository with latest commit from the remote repository : When you are working in your own local repository, the remote repository can have some new commits. To have your local repository updated with changes (i.e new commits) from the remote repository, you can use the *git fetch* command. This command fetches remote commits into your local repository as a 'remote' branch and updated the 'remote' reference object to point to the latest commit of this 'remote' branch. The 'branch' reference object still points to the latest commit of your 'master' branch. Another option is to use the *git pull* command. This command not only fetches remote changes but also 'merges' them into the current 'master' branch.

— *git fetch* : downloads changes from a remote repository into the local clone and updates the 'remote' reference object but does not merge them into the current local version.
— *git pull* : fetches remote changes into the local clone, and merges them into the current working files.

- Publish your change to the remote repository : When you want to share your work in a branch with others, you need to push it up to the remote repository. Note that you need the 'write access' to push directly to a remote repository. An alternative way to share your work is to use 'git request-pull' followed by 'git format-patch' and 'git send-email' to pass your work (patches) to the project maintainer of the remote repository.

— *git push* : uploads changes from local branches to the respective remote repositories.

To wrap up this subsection, we go through an intuitive example in which we clone a remote repository to local machine, edit and commit to local repository, fetch changes from the remote repository and merge them with our work on the local one. Figure 2.8 illustrates 'getting a

repository' by using the *git clone* command. The whole remote repository including files and git data (i.e the version history) is downloaded to the local repository. To simplify the example, the remote repository has only two commits in which the latest commit is 'RC1'. In the local cloned repository, we have also two commits and two reference objects. Both the 'remote' and the 'branch' reference objects point to the latest commit 'RC1'.

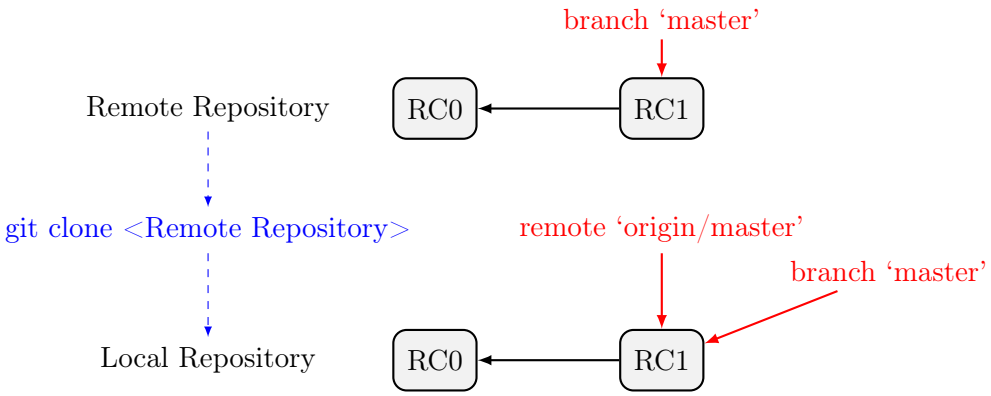


FIGURE 2.8 – Clone a repository

After getting a repository to local, we can start editing locally (i.e in isolation with the remote repository). Supposing that we save a new commit 'L1' to our local repository. Only the 'branch' reference object (branch 'master') is changed. The 'remote' object is not changed even if there is some change (i.e new commit) in the remote repository. Figure 2.9 illustrates the case in which both remote and local repository have a new commit. The 'remote' reference object in the local repository is not changed unless we run the *git fetch* command.

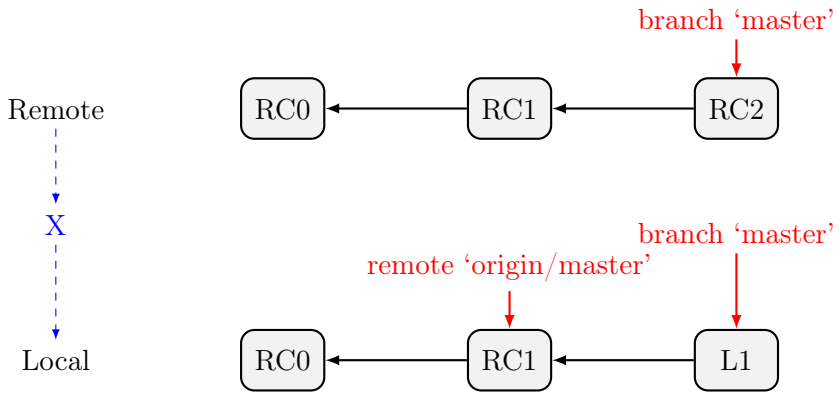


FIGURE 2.9 – Both repositories have new commit

Figure 2.10 presents the local repository after the *git fetch* command is run. New commit 'RC2' from the remote repository is downloaded to the local repository and the 'remote' reference object ('origin/master') is updated to point to that new commit. At this point, the local repository has all commits from the remote repository but its local commit L1 is not 'pushed' to the remote repository. There are two scenarios here. In the first scenario, our local work is already finished and we want to merge them into the remote main line, i.e the remote 'origin/master'. To do this, we need to perform a merge between two commits 'RC2' and 'L1' and then push our local changes to the remote repository. Note that conflicts can happen when we perform the

merge and we need to resolve them manually before pushing them to the remote. In the second scenario, we did not finish our local work yet and do not want to merge them back to the main line. We then can keep the two branches separately as long as we want. Note that longer the two branches are kept separately, the more conflicts we get when merging them later on. We can also share our local branch 'master' with other people by publishing (i.e 'push') them to the remote repository. Other people then can download and edit them on their local repository.

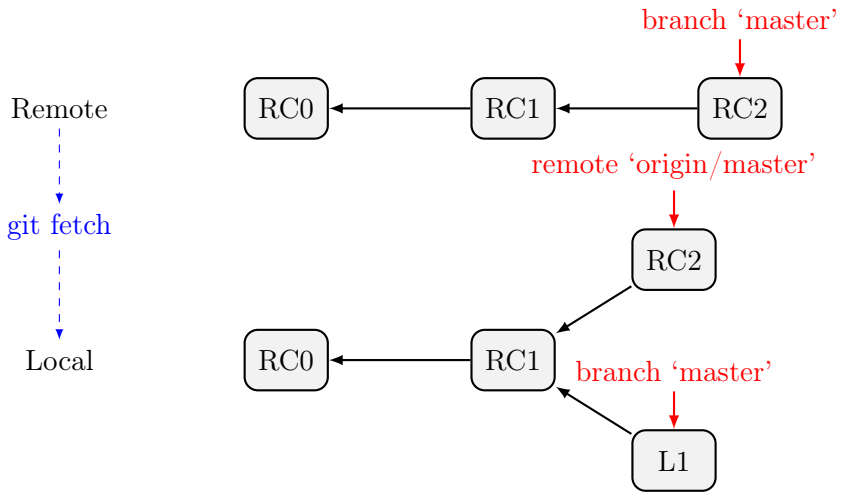


FIGURE 2.10 – Fetch changes from remote repository

Figure 2.11 shows the result of local and remote repository after merging local changes with remote changes and pushing the result back to the remote repository. Suppose that 'L1' and 'RC2' are merged into 'M1' successfully. Now the remote repository also has 'L1' (local work) and 'M1'(merge result) commits.

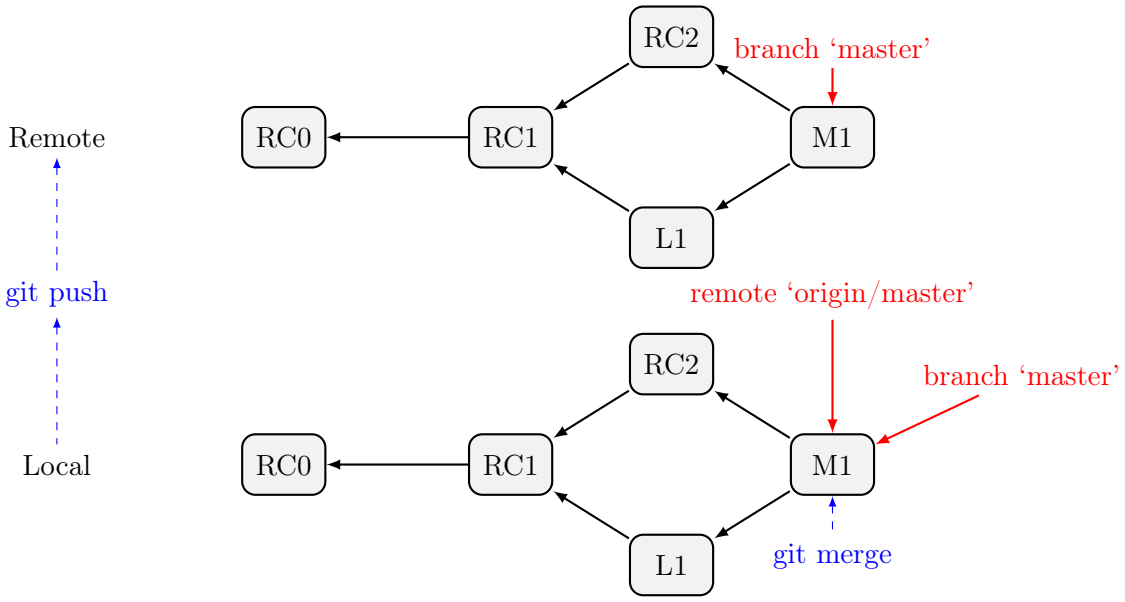


FIGURE 2.11 – Merge and Push local changes to remote repository

Practical models and Git workflows

The above intuitive example is a very simple use-case which illustrates the basic workflow in Git. However, in the real software projects, especially the large-scale open-source projects such as Linux Kernel [35] or Ruby on Rails [32], the scenario is more complex as we have many collaborators (developers) and many concurrent activities. For instance, Figure 2.10 presents another example in which the remote repository has two branches : 'master' and 'feature'. The local repository has two 'remote' reference objects respectively. When we fetch remote changes to the local repository, the local repository is updated from these two branches. If we want to merge our local branch 'master' to the remote repository, we need to merge the local branch 'master' with the two remote branch 'origin/master' and 'origin/feature'. In this case Git provides an 'octopus' merge strategy which merge multiple branches into a new commit. This merge strategy is useful for the case in which user needs to merge many 'feature' branches into the 'master' branch. However, the 'octopus' merge strategy does not support resolving conflict manually. If the merge results in conflicts, Git is going to cancel the merge and user needs to use another merge strategy instead of 'octopus'. The 'octopus' merge is more likely to succeed if these 'feature' branches do not share any common files.

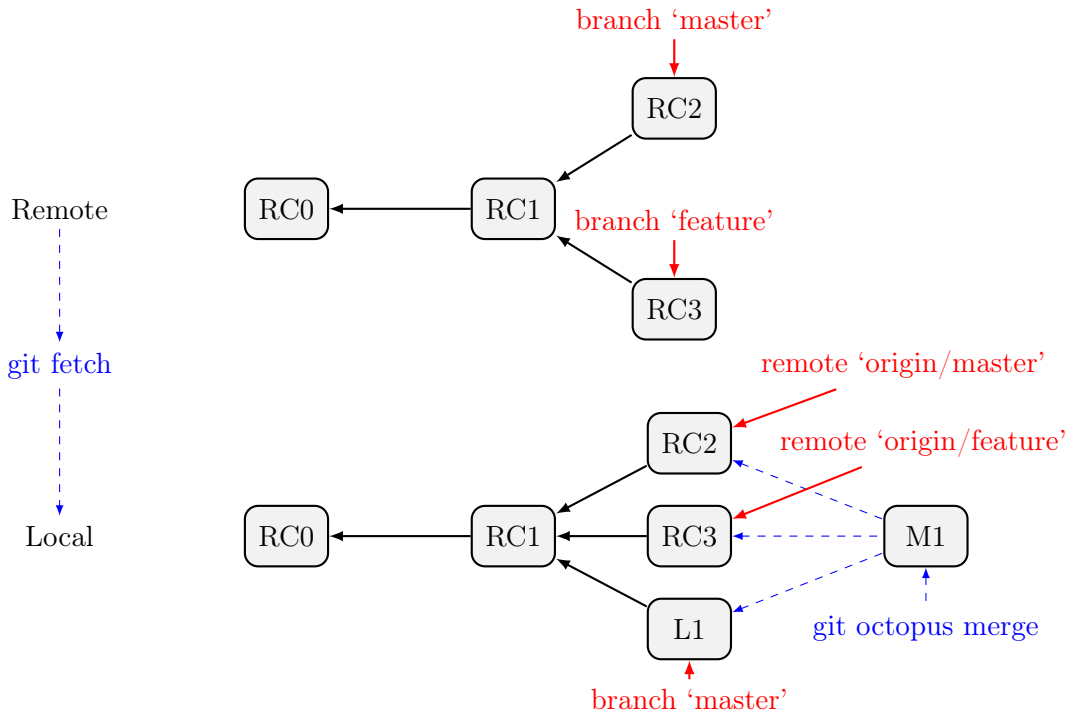


FIGURE 2.12 – Remote changes include multiple branches

Another problem is the process of 'publishing your works to a remote repository' which requires the 'write' access right on the remote repository. It's quite complex if the remote repository is stored in a personal computer/laptop of another person. Even if the remote repository is stored in a server which can be reached via http/ssh protocol, only few people have the 'write' access right. It means that only few people can 'push' their works directly into a remote repository. To deal with this problem, the 'pull-request' model which is based on the 'pull' operator of Git has been developed. It's also called 'pull-based' model as the 'push' operator is not allowed in this model [46] . Collaborators can clone a remote repository. However, when they want to publish

their work into the remote repository, they need to ask its Owner to 'pull' their work into the remote repository. In addition, not only the official repositories that are stored in the server, any collaborators can make their local repository available on the internet by using web-based Git hosting services such as GitHub [47] (free) and Bitbucket [48] (commercial/free with limitation). These Git-hosting services create an 'interface' repository which is synchronized with user's local repository. The synchronization process needs to be done manually by users. Actually it's the process of 'push' changes from local repository to its public 'interface'. Any 'clone', 'pull' or 'push' actions from other repositories are performed on this public 'interface' of the local repository. Figure 2.13 illustrates how two private repositories can interact with each others through their public 'interface'. To simplify the case, we consider the local repository and its public 'interface' as one. The synchronized phase between them is transparent to other users.

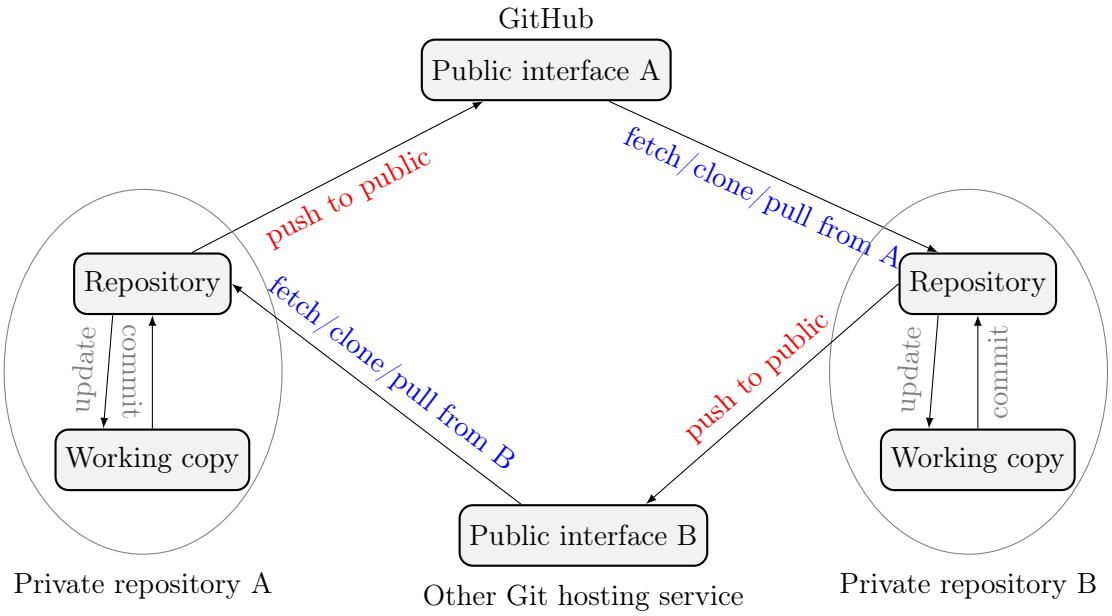


FIGURE 2.13 – Git hosting services - Public interface of local repository

As we had introduced earlier, Git has a 'decentralized' architecture. Theoretically, there is really no special repository as all repository store files and also the commit history. You can download (clone) a project from any reachable repository of it. However, in real projects, there is always an 'official' repository (called 'blessed repository') where users can find new official updates and changes. In Git, beside the 'remote' reference object created when you clone a project, you can add many other 'remote' reference objects which 'link' to many other repositories. They can be the repositories of your close co-workers or the repository of a famous developer that you want to follow. In another way, Git allows 'multiple remote' ; you can pull from and push to multiple repositories.

- Central repository model : Normally in small and medium size projects, this 'blessed' repository works as a 'central' repository. All developers can 'clone' the project from this central repository. But only a group of core-developers who can push directly to this 'central' repository while all other developers need to use 'pull-request'. This is called 'central repository model which is illustrated in Figure 2.14. This model is also used in some large size project such as Samba [34] and IkiWiki [33]. In Samba, the 'central' repository is shared between 'registered' developers along with a set of tools supporting building and testing automatically. In IkiWiki, there is only

one ‘core’ developer, Joey Hess, who is also the creator of the project. All contributions are handled by him. Contributors need to create ‘patches’ from their work and send them to Joey.

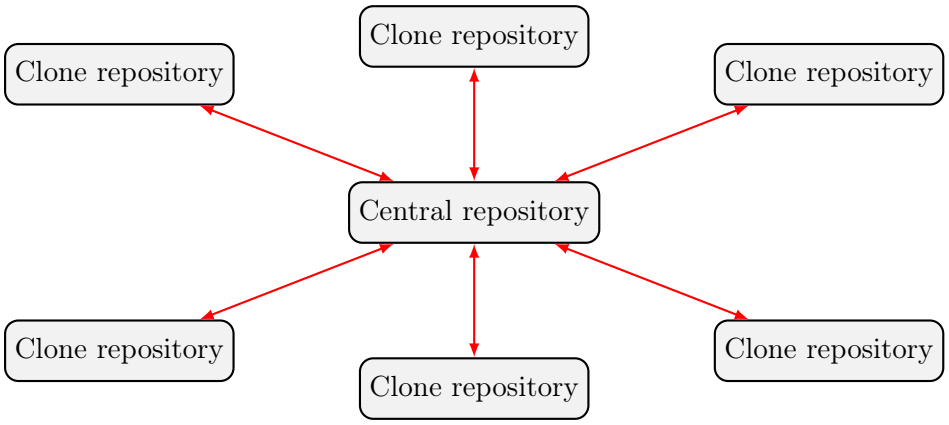


FIGURE 2.14 – Central repository model

- Hierarchical model with sub-maintainers : Merging is always a big problem when using ‘central repository model’ in a large team. ‘Hierarchical model’ is a more suitable model for medium and large size projects. In this model, all developers can clone or update their existing clone with new changes from a ‘blessed repository’. However, when they want to ‘push’ their work, they make a ‘pull-request’ to another ‘integration manager’ repository. Figure 2.15 presents a highly hierarchical model with sub-maintainers which is used in Linux Kernel project [45]. In this model, the ‘integration manager’ includes ‘lieutenant’ and ‘dictator’ layers. Depending on their works, collaborators make a ‘pull-request’ to different ‘lieutenant’. The manager of each ‘lieutenant’ pulls changes from developers, builds and tests at ‘lieutenant’ repository. The ‘dictator’ then pulls changes from the ‘lieutenants’, builds and tests before publishing them to the ‘blessed’ repository. The number of sub-maintainers depends on the size of each project.

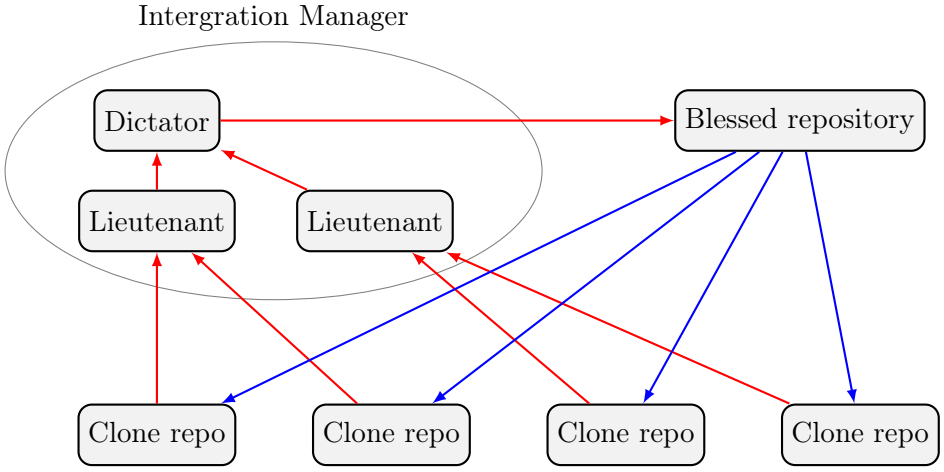


FIGURE 2.15 – Hierarchical model - ‘Dictator-and-Lieutenant’

Conflict merge and resolution in Git

'Branching and Merging', or 'Divergence and Synchronization', are the main actions of Git. When you get a repository by cloning another repository or initializing a new one, actually you get a local branch 'master'. You can also create a new branch with specified name easily by using *'git branch'* or *'git checkout -b'* commands. However, the difficult part is the integration phase in which we need to merge several branches into the 'master branch' (the main line). Git, like other version control systems, provides an auto-merging mechanic that can merge concurrent changes (i.e changes that are made in different repositories in parallel). In general, there are three types of merge : 'Fast-forward' merge, '3-way' merge and 'Octopus merge'.

- *'Fast-forward' merge :* this type of merge happens when a user is the first one since the latest update, who 'pushes' changes from local repository to remote repository. It's actually just a synchronizing of repository states, not a merge. Figure 2.16 presents an example of 'Fast-forward' merge. In this example, user clones the remote repository when 'RC1' is the latest commit. User then makes a new commit 'L1' and pushes changes to the remote. The merge between remote 'branch master' and local 'branch master' is actually the process of copying commit 'L1' from local and 'forward' the 'branch master' reference to the latest commit 'L1'. 'Fast-forward' merge always succeeds.

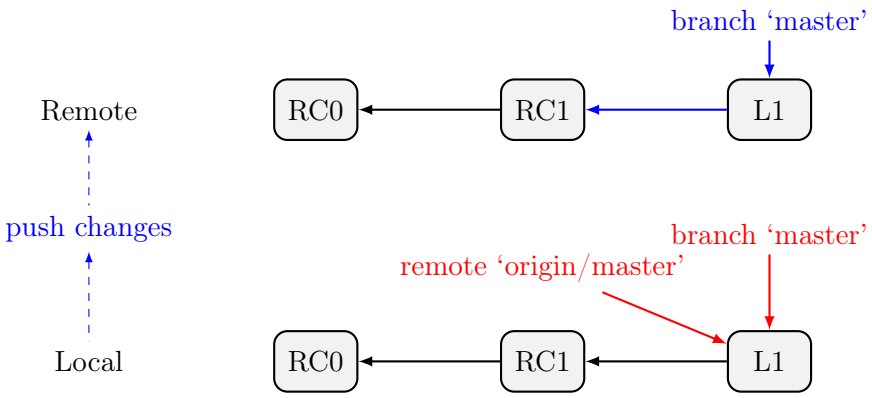


FIGURE 2.16 – Fast-forward merge

- *'3-way' merge :* Following up the example in Figure 2.16, if there is another commit 'RC2' which was pushed to the remote repository before 'L1', then user needs to merge 'RC2' and 'L1' before pushing 'L1' to the remote. In case of a successful merge, the result is similar to the example in Figure 2.11 in which the merge result 'M1' is created. To merge these two branches, Git takes into account also the common ancestor of them (i.e commit 'RC1'). If the 'Fast-forward' merge is the first merge since the latest update and always succeeds, the '3-way' merge may result in conflict. There are several types of conflict which can be categorized into three general cases : *'update-update conflict' (also called 'content conflict'), 'update-remote conflict' and 'naming conflict'* . In Git's logs, *'update-update conflict'* is marked as 'CONFLICT (content)' ; *'update-remote conflict'* includes 'CONFLICT (rename/modify)' and 'CONFLICT (rename/delete)' ; *'naming conflict'* includes 'CONFLICT (rename/rename)', 'CONFLICT (rename/add)', 'CONFLICT (modify/delete)' and 'CONFLICT (add/add)'.

- *'Octopus merge :'* As Git allows users to create several branches easily, it also provides the ability to merge more than two branches. However, Git supports only the simple merge, i.e merge without conflict. If there's any conflict during the merge process, Git will cancel the octopus merge and users then need to repeat using '3-way' merge to merge two branches at a

time. If you have several 'feature branches', you will need to merge them, one at a time, to the 'master branch'.

Conflict merge needs human efforts to resolve. If a 'Octopus' merge results in conflict, it simply requires user to use '3-way' merge instead of 'Octopus merge. Conflict in '3-way' merge is more complex to resolve. In Git, the merge of two branches (with referencing to their ancestor) is the merge of two latest snapshots of two branches of a project. And there are many files in a project. It means that when the merge results in conflict, we may have many conflicts. Git provides us a list of conflicts including related files and conflict types (*'update-update conflict', 'update-remote conflict' and 'naming conflict'*). The *'update-remote conflict' and 'naming conflict'* are simple to resolve. Users can choose to keep or remove a conflicting file or can just rename two conflicting files. The *'update-update conflict'* (i.e 'content conflict') is more complex to resolve, especially in the case that two users had edited them in isolation for a long time. In this case, Git tries its best to provide us a '3-way' conflict report including changes from both sites (branches) in comparison to the original version (the common ancestor of the two branches being merged). Note that this conflict report is stored in the new 'conflicting version' of the 'conflicting' file. Users can use *'git status'* command to see the list of 'conflicting' files which are marked as 'unmerged'. Figure 2.17 presents the conflict report of a simple conflicting file 'text.txt'. The 'text.txt' file has only two lines. The first line of it is changed in 'Branch 1' and also 'Branch 2'. The second line is kept unchanged in both branches. A '3-way' merge between Branch 1, Branch 2 and their ancestor obviously results in conflict. Git presents this conflict in *'diff3'* format which points out clearly which lines are changed, which lines are unchanged. For changed lines, Git provides both versions of change and the original version. It makes users resolve the conflict easily.

In addition, Git also allows users to cancel (or undo) a merge by using *'git reset'* or *'git revert'* commands. Users can cancel a merge in case it's difficult to resolve the conflict. In software development, merging a content conflict in a 'code file' (a text file written in a programming language such as Java or C++) is even more complicate. After fixing the *'textual conflict'* (i.e content conflict) of a merge, users can get problems/errors when compiling or running the code. This type of 'semantic conflict' (also called 'higher-order-conflict') requires more effort to be resolved [15]. Another case of semantic conflict that user may encounter after a merge, either success or conflict and fixed manually, is the case in which changes of a software artifact (a code file) may affect concurrent changes of another software artifact [49]. Users are more likely to 'roll back' when getting these complex types of conflict as it's very difficult to resolve them, especially the later case (also called 'indirect conflict').

The Commit History

Each Git repository has a commit history which stores all commits and merges related to the repository. Two repositories of a project may have some different commits, merges or branches but generally they have the same 'main line' (i.e the important commits, merges and branches). In addition, they often have a 'link' (a reference) to a common 'official repository'. Different projects may have different 'models' of using Git such as 'Central repository model' or 'Hierarchical model'. However, they always have an 'official' (or 'blessed') repository where they can pull updates from and publish their work to (using 'pull-request' model). It's obvious that the commit history of a local repository belonging to an individual is different from the one of the 'official' repository. Users can create several branches and commits in their local repositories and push a few of them which are important to the 'official'. The remaining commits and branches that they do not want to publish are unfinished things or they were just for testing purpose. Nevertheless, the commit history of this 'official' repository is the most adequate commit history among many

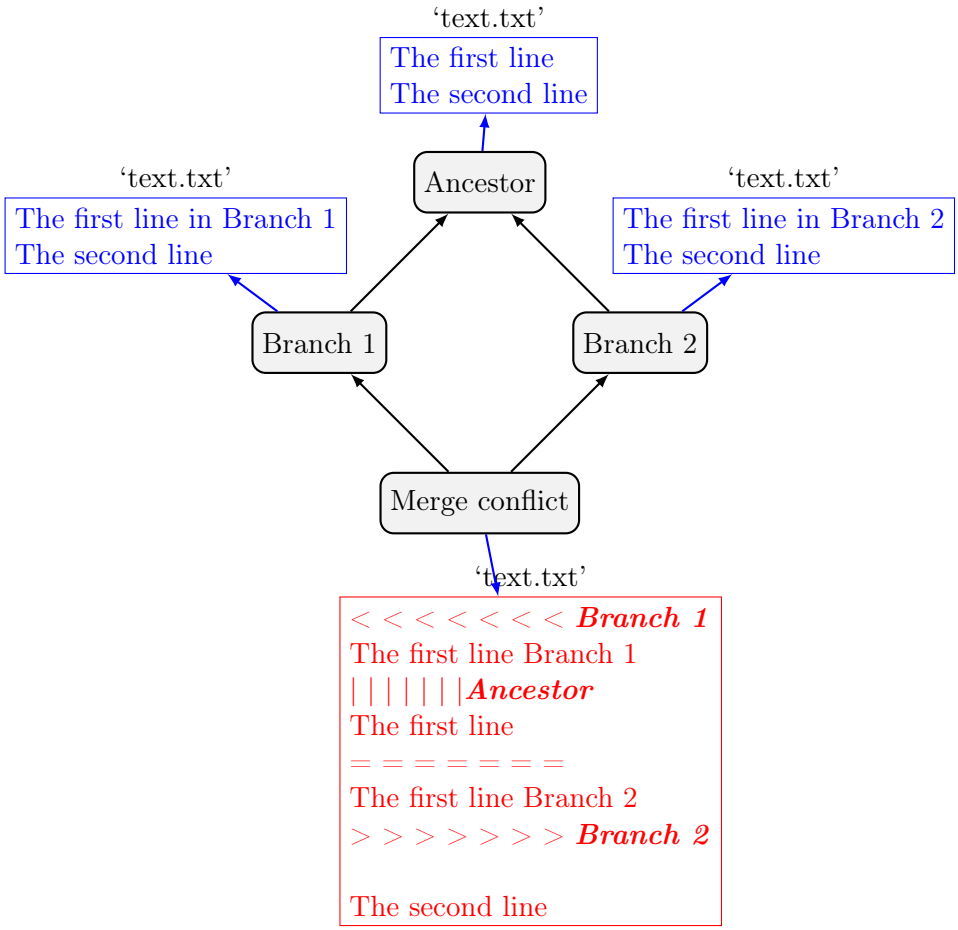


FIGURE 2.17 – Conflict report of file 'text.txt' [update-update conflict]

repositories of the projects. In open-source software community, the 'official' repository is usually set to 'public access', i.e everyone can get a 'clone repository' to their local machine and can browse the commit history of the project using *git log* command.

According to the report in January 2020 of GitHub [47], a very popular Git-hosting service (i.e providing Git-public interface), over forty million users use GitHub and over one hundred million repositories were created. This is a valuable corpus to study about how users (developers)employ Git to collaborate on software development. Bird et al [50] confirmed that *'DSCMs (i.e DVCSs) promise new and useful data to help us better understand software processes'*. However, the authors also outlined some 'pitfalls' on mining these DSCM data. Notably, the commit history can be re-written by developers. Of course, only developers who have the 'write access' can edit the commit history. A developer can use 'git rebase' command to simplify his commit history before publishing his work to a public repository. In Samba project [34], developers are encouraged to use 'git rebase' in their local repository before 'pushing' their work to the 'build' repository. It means that a developer can create or 'pull' different working-branches to work on his local repository. However, when he finishes his development, he can use 'git rebase' to simplify his commit history into a 'linear commit history'. The details of branching and merging in his local repository is deleted from the commit history. In general, developers are advised not to use 'git rebase' in a public repository [17].

2.1.3 Real-time collaborative editing

A collaborative editor is a Groupware (also called Collaborative software)belonging to the 'Multi-users editor' category in the Application-Level taxonomy of Groupware by Ellis et al [7] in the early of 1990s. As opposed to conventional editors which allow only one user to edit a document at a time, collaborative editors allow a group users to modify a shared document simultaneously. Nowadays, collaborative editors are usually a web-based application (such as Google Docs [22], Etherpad [24], ShareLaTeX [23]...) which allow multiple users to edit collaboratively in real-time. Edits from different users are merged almost immediately.

As editing operations performed by different users may conflict, collaborative editors need a protocol to ensure the convergence of the shared document. The two most popular synchronization algorithms are 'Operational Transform' (OT) and 'Conflict-free replicated data type'(CRDT).

'Operational Transform' (OT)

OT is often called as 'the real-time collaborative editing algorithm'. It is used as a core-technique in many popular online editors supporting collaborative editing such as Google Docs , Etherpad and ShareLaTeX. OT was first implemented by Ellis et al [25] in GROVE system (Group outline Viewing Editor). The authors named it as 'Distributed Operational Transformation' (abbreviated as 'dOPT'). The basic ideal of *dOPT* is make sure all generated operations have been executed at all sites while maintaining their 'precedence property'. For instance, if operation 'o1' precedes operation 'o2', then the execution of 'o1' always happens before the execution of 'o2' in all sites. *dOPT* algorithm uses a *'transformation matrix' (denoted as 'T')* as the key to resolve conflicting operations. Each site has a *'State Vector'* including 'N' components ('N' is the number of collaborative sites), a *'Request Queue'* which keeps requests waiting to be executed and a *'Request Log'* which maintains a log of requests executed at its site. Component ith of *'State Vector'* of site j indicates how many operations from site i have been executed by j. When an operation o is generated at a site i with current state vector s, the site executes operation o, generates a priority for operation o, p, and sends a *Request* under the form of '$<i$,s, o,$p>$' to all other sites. When site j receives a request '$<i$,s, o,$p>$' of operation o at site i, information of state vector s is examined to see if the sending site (site i) executed operations which have not yet been executed at received site (site j). If so, the operation is queued (i.e 'future operation') ; if not, the operation is executed. And if site j has executed operations which have not yet been executed at the sending site i (i.e 'past operation') then the priority of the operation is checked and the operation may be transformed before it is executed.

The *'transform function'* is the most important part of any OT algorithm to handle concurrent operations. In general, *'transform function'* generates a new operation from two operations that have been applied to the same state of a document but on different clients. *'Transform function'* is usually denoted as $T(a, b)$ in which operation a is transformed against operation b. The new generated operation $a' = T(a, b)$ can be applied at a client in which operation b has been applied and can still preserve the intended change of operation a.

'TP1' and 'TP2' are two transformation/convergence properties to ensure the correctness of the *'transform function'*. Figure 2.18 illustrates *'TP1'* property and Figure 2.19 illustrates *'TP2'* property.

'TP1'(transformation property 1) is also known as *'TP1/CP1'* (transformation/convergence property 1) which defines a 'state equivalence' : *The state generated by the execution* $o1$ *followed by* $T(o2, o1)$ *must be the same that the state generated by* $o2$ *followed by* $T(o1, o2)$. In more detail,

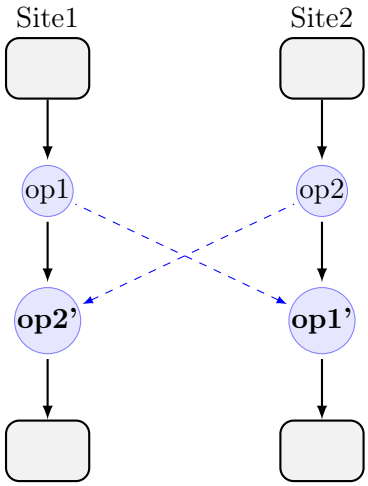


FIGURE 2.18 – TP1 property illustration : $o1 * T(o2, o1) \equiv o2 * T(o1, o2)$

for two concurrent operation $o1$ and $o2$, the transform function T satisfies *'TP1'* if and only if $o1 * T(o2, o1) \equiv o2 * T(o1, o2)$ where '*' denotes the sequence of operations. If the Operational Transformation systems allows any two operations to be executed in different orders, it needs to satisfy *'TP1'* to achieve convergence.

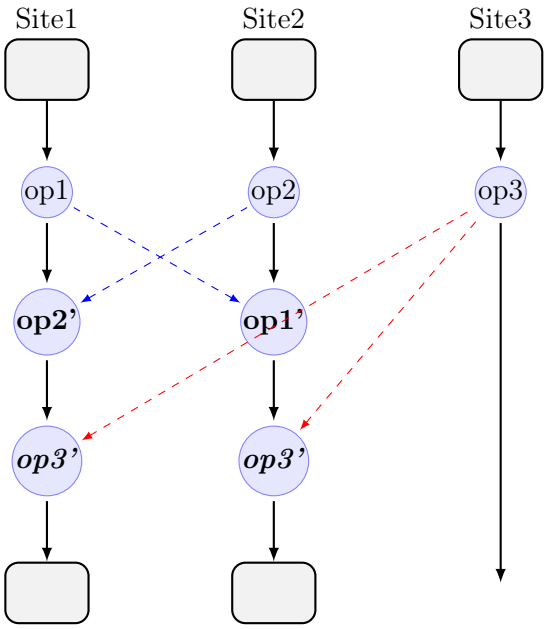


FIGURE 2.19 – TP2 property illustration : $T(o3, o1 * T(o2, o1)) \equiv T(o3, o2 * T(o1, o2))$

'TP2' (transformation property 2) is also known as *'TP2/CP2'* (transformation/convergence property 2) which ensures that : *the transformation of an operation to a sequence of concurrent operation does not depend on the order of the transformation of that sequence.* In more detail, for three concurrent operations $o1$, $o2$ and $o3$, the transform function T satisfies *'TP2'* if and only if $T(o3, o1 * T(o2, o1)) \equiv T(o3, o2 * T(o1, o2))$. *'TP2'* is required if the Operational Transformation systems allows any two operations to be executed in two different document states (contexts).

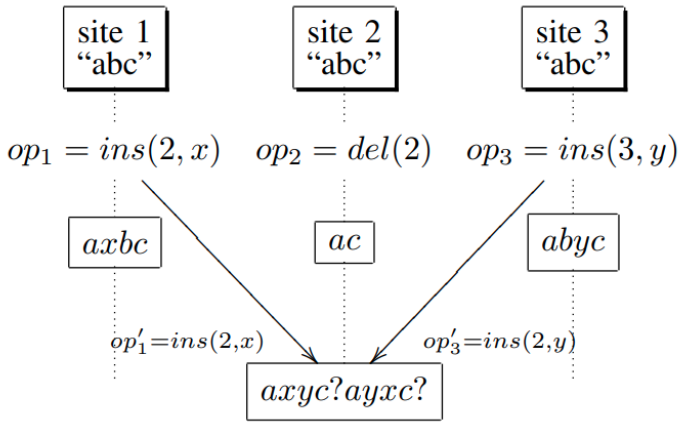
OT is a powerful tools that allows to build great collaborative apps without the need of blocking concurrent editing. It has had many implementations over the years. Known as the first OT algorithm, *dOPT* enforces the *'TP1'* property. However, it fails when an operation is concurrent with two or more dependent operations.

adOPTed [51] introduced the second transformation property, *'TP2'* property, which ensures that transformation of an operation along different paths will get the same result. *adOPTed* algorithm solves most of the limitations of *dOPT*.

Sun et al [52] presented an optimized version of generic operational transformation, *GOTO* algorithm. The algorithm achieves the convergence property (TP1 and TP2), causal order and has 'undo' operation. However, its correctness is not theoretically proven for all scenarios.

Google Wave OT algorithm is basically based on *Jupiter* algorithm (Jupiter System) [53]. In *Google Wave OT*, the client need to wait for acknowledgement from the server before sending more operations. When waiting for acknowledgement, the client caches its operations (local operation) and sends them later. When the server has transformed the client's operation, applied it to the server's copy and broadcast the transformed operation to all other connected clients, it then acknowledges the client.

The above is some proposed 'state of the art' OT algorithms. Fulfilling two transformation/convergence properties *'TP1' and 'TP2'* is the key to ensure the correctness of an OT algorithm. However, satisfying the *'TP2'* property is very difficult. Imine et al [54] had proved by 'counter example' that almost all proposed OT algorithms do not satisfy the *'TP2'* property. A solution proposed to this problem is to require only the *'TP1'* property and implement the unique global serialization order (continuous order) such that the operations can be delivered in this order [55]. Vidot et al implemented *SOCT4* using this strategy. In *SOCT4*, the operations are ordered using a timestamp given by a sequencer to achieved a global continuous order. The delivery and integration of remote operations can be made depending on the given timestamp of each remote operation. Besides, concerning the broadcast of an operation, *SOCT4* deferred the broadcast of an operation until all preceding operations in the global continuous order have been received. *SOCT4*, however, requires a central site to build the global continuous order. It prevents *SOCT4* to be used in a peer-to-peer (pure decentralized) network.



Source: [56]

FIGURE 2.20 – Common problem of OT algorithms.

A common problem of all proposed OT algorithms is showed in Figure 2.20. In this figure, three concurrent operations *ins(2,x), del(2) and ins(3,y)* happen in three different sites *site1, site2*

and site3 respectively. Considering the transformation on *site 2*, when $op1 = ins(2, x)$ arrives, it is transformed according to $op2$, we have : $op1' = T(op1, op2) = ins(2, x)$. This transformation keep the original insertion position of $op1$ because it is smaller or equal to the deletion position of $op2$. Similarly, when $op3$ is received on *site 2*, it is transformed based on $op2 : op3' = T(op3, op2) = ins(2, y)$. The insertion position of $op3$ is decreased (by '1') because it is larger than the deletion position of $op2$. At this point on site2, $op1' = ins(2, x)$ and $op3 = ins(2, y)'$ have the same insertion position. The transformation of $op1'$ according to $op3'$ $((op1')' = T(op1', op3')$ and the transformation of $op3'$ according to $op1'$ $((op3')' = T(op3', op1'))$ give different results (i.e $axyc$ or $ayxc$). Unfortunately, almost all proposed OT algorithms fail to order correctly $x(op1')$ and $y(op3')$ (all the counter-examples in [54] are instances of this tie). The correct order is to insert x before y [56].

Oster et al [56] proposed the 'Tombstone Transformation Function' (TTF) to maintain consistency in collaborative editing systems based on Operational Transformation algorithms. Tombstones Transformation Function (U-TTF, D-TTF) satisfy *'TP1' and 'TP2'* properties and also preserve intention of operations. In this approach, deleted characters are kept as tombstones instead of being removed completely. For instance, if a character of a string is deleted, it is still kept in its position in the string and marked as 'invisible'. In another way, 'deleted' characters still remain in the 'model' of the string but are hidden from the 'view' of the string which is seen by users. Figure 2.21 illustrates how 'TTF' (U-TTF) is applied to the 'common problem' example. In this example, b separates x and y and as 'b' is not actually 'deleted', the algorithm can generate the correct order of x and y. The system 'model' keeps deleted character '~~b~~' at its position and inserts x, y before and after '~~b~~' respectively. In another way, operation $op3 = ins(3, y)$ is transformed into $op3' = ins(3, y)$ according to operation $op2$, then transformed into $(op3')' = ins(4, y)$ according to operation $op1'$. In case the algorithm wants to transform $op1'$ according to $op3'$, $(op1')' = T(op1', op3')$ is still $ins(2, x)$. However, the execution of $(op1')'$ will 'shift' [~~b~~, y, c] to one step to the right and we have the same result as transforming $op3'$ according to $op1'$ (i.e y is at position 4 and x is at position 2).

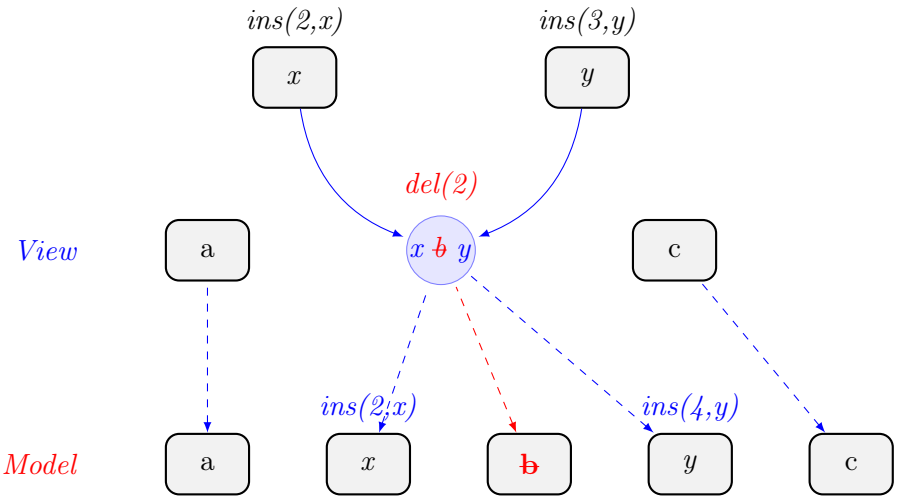


FIGURE 2.21 – Applying TTF (U-TTF) for the 'common problem' example

'Conflict-free replicated data type'(CRDT)

CRDT is considered as the newer approach to real-time collaborative editing. In contrast to OT, this approach does not take the sequence of operations as the key of synchronization. Instead, it considers the synchronization in term of underlying data structure. *CRDT*, at the general level, is an object type of which objects of the same type can be merged in any order to produce an identical union object. The replicas of a *CRDT* converge automatically. In collaborative editing, *CRDT* provides the way to merge concurrent changes in any order and always satisfies consistency preservation. In *CRDT*, changes are split into such small pieces that they do not need to be transformed in generally, but only their positions are changed. For textual changes, every character is treated as a separated entity (*CRDT* supports collaboration on two main types of data : plain text and JSON structures).

There are two types of *CRDT* : operation-based CRDT (also called 'commutative replicated data type', CmRDT) and state-based CRDT (also called 'convergent replicated data type', CvRDT). State-based CRDT is simpler to implement but it's costly as the entire local state of each site is transmitted to other replicas. Operation-based CRDT, in contrast, transmits only the updated operations. However, operation-based CRDT requires to guarantee that operations are not dropped or duplicated and that they are delivered in causal order.

CRDT was first mentioned as the data structure in WOOT (WithOut Operational Transformation) framework [26]. In WOOT, each character has a unique identifier, and maintains the identifiers of the previous character and following character at the initial execution time. WOOT does not require any vector clock contrary to most operation transformation based algorithms. It also does not require a primary site. WOOT, however, retains tombstones to record deleted characters. Keeping tombstones helps the system to enable 'group undo' any operations but it makes the space overhead increase significantly.

In another approach, Preguica el tal [57] proposed 'Treedoc', a CRDT solution for sharing edit buffer based on an extended binary tree. This extended binary tree is used for building unique identifier (with and without tombstones) with required properties. The tree is optimized to avoid to be unbalanced which may cause overhead.

Shapiro et al [27] formally presented the concept of a *CRDT* for both state-based and operation-based styles. They presented also some simple *CRDT* examples such as 'integer vectors and counters', 'U-set, map and log' and detailed about 'directed CRDT graph' which might be used in large-scale web search engine.

Some popular Web-based Real-time collaborative editors

Many web-based real-time collaborative editors have been developed. Some of them are used only for experimenting purpose and also some of them became very popular editing tools. In this sub-section, we describe quickly three popular web-based real-time collaborative editors. They are all based on centralized architecture and Operational Transformation algorithm. Besides, we also introduce MUTE which is one of the rare collaborative editors that relies on peer-to-peer architecture and Conflict-free replicated data type.

Google Docs [22] is a cloud (online) document editor in the Google cloud office suite (including Google Docs, Sheets, Slides and Google Forms). Google Docs, like other applications in the office suite, supports multiple users editing collaboratively in (near) real-time. Changes (edits) from users are saved to Google server automatically whenever they are made. A revision history is kept. It allows users to see changes made to a documents, to compare two adjacent revision of a documents. Note that users can not control how often revisions are saved. Google Docs relies on

Jupiter algorithm based on Operational Transformation to solve conflicts in collaborative editing process automatically.

Etherpad [24] is a centralized web-based real-time collaborative editor that support only plain-text documents (called 'pad'). To edit collaboratively, users need to use a web browser to connect to a Etherpad URL (i.e Etherpad server). When a user opens and edits a 'pad', local changes are sent to a centralized server and broadcasted to all other connected clients in (almost) real-time. Etherpad uses Operational Transformation algorithm to maintain its editing consistency.

ShareLaTeX [23] is an online LaTeX Editor supporting collaborative editing. ShareLaTeX borrows some ideas from version control systems(such as Git), it allows users to save 'labelled versions' which are like commits. You can always get back to and compare with other versions. You can also access to your ShareLaTeX project via Git. ShareLaTeX use Operational Transformation to manage concurrent edits. Edits on ShareLaTeX are sent to the server every few seconds. If two or more people edit in parallel, the server can 'rebase' their change so that all clients end up at the same version. The server uses 'web sockets' to push updates to all connected clients. In July 2017, ShareLaTeX joined forces with Overleaf.

MUTE (Multi-User Text Editor) [58] is a peer-to-peer web-based real-time collaborative editor that relies on 'CRDT'. MUTE provides a peer-to-peer collaboration which supports both online and offline work modes (i.e 'real-time' and 'asynchronous'). In contrast to other systems that are based on a central authority, MUTE allows users to maintain their data and select co-authors with whom they share it.

2.2 Studies on Collaborative work based on version control systems

Version control is often used in software development. It allows developers of a team to work in collaboration in a software project and also brings many other advantages. A project manager can easily see who made changes and what those changes encompassed. If the current version of a project contains some errors that are complex and time-consuming to be solved, the project manager can consider to roll-back to the latest stable version as a quick solution. Besides, version control systems provide the complete history of all versions and the differences between two selected versions of a file or a group of files. Version control does not only support well collaborative work in software development but it is also useful for other works that deal with changes on file/files overtime, especially including multiple collaborators.

Some early researches on working model (i.e work mode) for Collaborative editing were based on version control systems. In the early of 1990s, Minor and Magnusson [6] presented a model for semi-(a)synchronous collaborative editing which heavily relies on version control to manage changes on hierarchically partitioned documents. In their model, a document is partitioned into many sub-parts such as chapters, sections and paragraphs in a text document or modules, functions and classes in a code (programming) document. These sub-parts are subjected to the version control system instead of the whole document. When a user edits and submits a new version of a sub-part of a shared document, it also creates a new version of the document which contains the new version of edited sub-part and shares unchanged parts with the old version of the document. In addition to handling *versions*, this hierarchical fine-grained version control also supports collaboration awareness for collaborative editing in two forms : shared evolution graph and 'active-diff' presentation. The shared evolution graph allows users see the current status of a document such as who is editing, what has happened between different versions. The

'active-diff' gives the differences between two selected 'versions'. If one of them is editing while the 'active-diff' is active, the 'active-diff' will show the differences continuously.

In another approach, Dourish [11] focused on the *'divergence management'* in systems supporting collaborative work, especially in collaborative writing system. The author figured out that the *'inconsistency avoidance'* approach of distributed systems is not suitable to collaborative systems. In collaborative systems, the 'key distributed entities' are users who are not well-prepared to follow many constraints to avoid 'inconsistency'. The author then proposed the *'divergence management'* approach, which looks to manage the 'divergence' of *'multiple, simultaneous streams of activity'*. The process of collaboration is the *'splitting and merging'* or the *'divergence and synchronization'* continually of the streams of activities. Only at the point of synchronization, all streams have the same 'view data' , i.e all users have the same view. Further editing activities make them to diverge again, i.e users have different views, until the next synchronization point. This process of continual divergence and synchronization is similar to what version control typically supports. Version control systems maintain a historical versions of all objects (files) of a project. They allow multiple versions of an object to exist at the same time. And different versions of an object can be edited further and kept divergent(branching) or synchronized (merging). However, the version controls primarily emphasize on the creation and management of parallel versions within an 'asynchronous' context, i.e the 'divergence' status appears more often than the 'synchronization' status.

Beside defining a working model for collaborative work based on version control system, researchers analyzed the traces collected from collaborative work. Reiher et al [12] presented a study of conflict resolution in Ficus optimistic file system. Perry et al [13] studied about parallel changes in a subsystem of the Lucent Technologies' 5ESS over the period of twelves years. Zimmermann [14] analyzed the 'integration rate' and 'conflict rate' of four large open-source projects based on CVS (Concurrent Version System). Brun et al [15] and Kasim et al [16] studied the 'merge conflicts' in several open-source projects based on Git, a decentralized version control system.

The user study presented in [12] reports on conflict resolution experiences with the optimistic file system Ficus. Conflicts were classified into *update/update*, *remove/update* and *naming* conflicts. *Update/update* conflicts appear when two concurrent updates are performed on the same file. *Remove/update* conflicts appear when an update of a file and the removing of that file were performed concurrently. A *naming* conflict occurs when two files are independently created with the same name. The study found out that only about 0.0035% of all updates made to non-directory files resulted in conflicts and among them less than one third could not be resolved automatically. Authors mentioned that conflicts that cannot be resolved automatically are any *update/update* concurrent changes on source code or text files as they have arbitrary semantics and therefore require user intervention. Note that in contrast to the definition of conflicts used in [12], in the terminology of version control systems two updates done on the same file (source code or textual) lead to non-automatically resolved conflicts only if the updates refer to the same or adjacent lines in the file. All *update/remove* conflicts required human intervention and about 0.018% of all *naming* conflicts led to name conflicts which have to be resolved by humans.

In [13], the authors presented a study about parallel changes in the context of a large software development organization and project. The study analyzed the complete change and quality history of a subsystem of the Lucent Technologies' 5ESS over a period of 12 years. Each set of change requests representing all or part of a solution to a problem was recorded by the system. When a change from this set was made on a file, the system kept track of the lines added, edited or deleted. This set of changes composes a *delta*. It was found that 12.5% of all *deltas* were made to the same file by different developers within a day. 3% of all these *deltas* made within a day

by different developers physically overlap. However, interference of these *deltas* is analysed over a quite large period of time (1 day) and not all these *deltas* are performed concurrently.

In [14] authors investigated four large open-source projects (GCC, JBoss, JEdit and Python) and found that in CVS the *integration rate* that measures the percentage of concurrently modified files over all modified files is very low (between 0.26% and 0.54%). The study found that the *conflict rate*, i.e. the proportion of files with unresolved conflicts over concurrently modified files, varies between 23% and 47%. Low integration rates indicate that the parallel changes within single files are rare and have small impact to the development process. High conflict rates indicate that parallel changes affect the same location within a file or can not be integrated automatically by CVS.

In [15] and [16], authors studied the *merge conflict rate* of merges for several open-source projects using Git repositories. They found that the average merge conflict rate was 16%. However, these studies did not analyse the types of conflicts during merge and the frequency of conflicts at file level as measured in [14]. Also these studies did not analyse quantitatively what are the resolution mechanisms adopted by users.

In [59], Brindescu et al analysed how centralised and distributed version control systems influence the practice of splitting, grouping and committing changes. However, this study did not analyse merge commits that represent decisions on conflict resolution. Our study is mainly focused on studying conflicts and therefore we analysed developers merging behaviour through merge commits.

In [60] authors propose a qualitative study on the factors that impact how practitioners approach merge conflicts and the difficulties they face when resolving conflicts. The study was conducted based on semi-structured interviews on 10 software practitioners across 7 organizations. The study found that the factors that mostly describe merge conflict difficulty are complexity of conflicting lines of code, the knowledge/expertise in area of conflicting code, the complexity of the files with conflicts and the number of conflicting lines of code. Our study is complementary to [60] and studies quantitatively textual conflicts inside a file and investigates different aspects such as their frequency, their length and their localisation and the ways developers resolve those conflicts such as by maintaining concurrent changes or rollbacking to previous code versions. It is important to understand what types of conflicts arise at the different moments during the lifetime of a project. This can help tool designers and developers to choose the different tools and techniques that can be applied during the lifetime of a project.

Despite the decentralized architecture, open-source software projects based on Git always have an 'official' public repository where developers can find the latest updates and releases. These 'official' repositories are the most important sources for researchers to retrieve the traces of collaboration. As many Git hosting services are available (i.e providing a 'public interface' for a repository), developers can make their own repositories publicly accessible such as the 'official' repository. Developers may have different content in their repositories. It's possible to recover more development histories (i.e the traces/logs of development process) from these repositories including the work that may never be integrated into the 'official' one. Based on these hosting service, German et al [61] presented an approach called *'Continuous mining distributed version control systems' (continuousMining)* that provides the ability to collect commit histories that are not included in the 'official' repository. The idea is to build a 'Super repository' that pulls/fetches changes (edits) from a set of all related repositories. This set is initialized with the 'official' repository and some well-known 'clone' repositories. The 'Super repository' is updated automatically with a fixed time interval t. If a new 'clone' repository is detected during an updating process, it will be added to the set of related repositories. The next update will pull changes from this new repository. The authors then apply *'continuousMining'* on Linux-Kernel project and found that

the 'Super repository' contains significantly more editing activities than from the 'official' repository. In a similar approach, Shapiro et al [27] build an 'Umbrella repository' that pulls/fetches changes/commits from all 'fork' repositories of an 'official' repository of a project. They then classified commits into different categories such as : 'UNIQUE commit' which exists only on one 'fork' repository, 'VIP commit' which exists in several 'fork' repositories (but not all) and the mainline (i.e official) repository, 'U-VIP commit' which exists in the mainline repository and only in one 'fork' repository, 'SCATTERED commit' which exists in several 'fork' repositories but not the mainline repository and 'PERVASIVE commit' which exists in all repositories.

As people are free to edit in their own local copy, Git needs a merge mechanism to integrate concurrent changes from different users. Perry et al [13] pointed out the need of supporting software merging during large-scale software development in distributed context. Tom Mens in his 'State-of-the-Art survey' on software merging [20] had provided a comprehensive overview of various merge techniques and categorized them according to their 'dimensions' : two-way or three-way merge ; textual, syntactic, semantic or structural merge ; state-based, operation-based or change-based merge. Mehdi et al [62] evaluated Software merge quality. The authors employed the publicly available history of six open-source projects to measure and evaluate the quality of merge tool results. Niemimen [63] proposed a process for real-time collaborative resolving of merge conflicts and also a web-based tool supporting this process in Git version control system. The process needs all participants, including the person who encountered the conflict and the 'helper' who usually has knowledge about the conflict, to be 'available' during the resolution.

In addition, distributed version control systems had brought a new paradigm for software development. Instead of using the 'copy-modify-merge' paradigm as in CVCSs, developers in DVCSs 'pull' and 'merge' changes from other repositories to their local repository. This 'pull-based' model is well supported by various hosting service such as GitHub for Git version control [46]. Gousious et al [64] built GHTorrent system which monitors 'public event time line' of Github. Each event is collected and stored to a MongoDB database which is a valuable corpus for analyzing how GitHub service supports 'pull-based' model in software development. Di Cosmo et al [65] presented Software Heritage project which collects, preserves, and shares the entire corpus of publicly accessible software source code. While GHTorrent system covers only GitHub, Software Heritage project covers also other software distribution places such as GitLab,SourceForge, Bitbucket. According to the literature, Software Heritage is the largest existing public archive of software source code with more than five billion unique source code files and one billion unique commits, coming from more than 80 million software projects [66].

Several studies of collaborative works based on version control systems have been conducted. Most of above studies focus on analyzing conflicts and resolution of concurrent changes in collaborative editing (specially in software development for these case). The reason of this interest can be explained by the fact that conflicts have significant effect to collaborative works (i.e development) as they are costly to be resolved and they can delay the collaboration work [67]. However, only few studies focused on analyzing conflicts and resolutions in DVCSs such as Git. They do not analyze conflicts at fine-grained level inside files and also do not take into account the different between CVCSs and DVCSs. For instance, a merge conflict in Git (DVCS) is a conflict at 'project' level, it contains one or many conflicted files. Besides, how people resolve merging conflicts is not analyzed in previous studies.

1. Team or group plans and outlines. Each member drafts a part. Team or group compiles the parts and revises the whole.
2. Team or group plans and outlines. One member writes the entire draft. Team or group revises.
3. One team member plans and writes draft. Group or team revises.
4. One person plans and writes the draft. This draft is submitted to one or more persons who revise the draft without consulting the writer of the first draft.
5. Team or group plans and writes draft. This draft is submitted to one or more persons who revises the draft without consulting the writers of the first draft.
6. One member assigns writing tasks. Each member carries out individual tasks. One member compiles the part and revises the whole.
7. One person dictates. Another person transcribes and revises.

FIGURE 2.22 – Seven organizational patterns, 1990, Ede and Lunsford[1]

2.3 Studies on Collaborative editing using real-time web-based collaborative editors

Since the late 1980s and early 1990s, Collaborative Editing became a major focus in Computer Supported Cooperative Work. Many collaborative editors had been developed and a number of studies were conducted.

Ede and Lunsford [1] presented a study of collaborative writing practices in seven professional associations which initial survey included 1400 randomly selected members. With 700 survey respondents, who in general spend 44% of their professional time on some kind writing activity, they reported that 87% of them 'sometimes write as members of a team or a group' and 58% of them agree that it is more productive of 'group writing' in comparison to writing alone. In which 'Group writing' was defined as '*Group writing includes any writing done in collaboration with one or more persons*'. The authors then did a second survey of twelve members of each organization which focused to describe the advantages and disadvantages of collaborative writing. Beside its advantages such as : joint knowledge, variety of approaches and ideas, different perspectives that generate better ideas for a better product and more accurate text ; they found that the disagreements in a 'common writing style' among individuals who don't want to change their own style is the recurrent problem in collaborative writing. The second problem, according to the respondents, is that group writing requires significantly longer time than individual writing. The third problem is the equitable division of tasks. And finally, 'group writing' can result in losing personal satisfaction and sense of creativity.

Further more, the authors described seven organizational patterns (Figure 2.22) and asked how often their respondents use them. The responses showed that 72% of their group writing followed a organizational plan and 95% of them perceived the need and productiveness of using writing plan. They also pointed out that mentoring ability and leaderships are 'characteristics of effective collaborative writers'.

Posner and Baecker [3] presented a taxonomy of joint writing based on the interviews of people that had participated in several collaborative writing projects. The taxonomy is composed of four categories presenting different perspectives of Collaborative editing : roles, activities, control methods and writing strategies. The 'roles' and the 'activities' focus on the users (i.e

collaborators) and their actions, the 'control methods' focuses on the document object and 'writing strategies' focus on how the writing process is managed and coordinated. Based on the taxonomy, the authors suggested a set of requirements for system supporting Collaborative editing.

Kim et al [10] interviewed 11 peoples who are working in academia about how they write together in practices. They focused on how people comment, edit and manage revisions in the shared document with existing tools.

Lowry et al [4] extended the taxonomy presented previously by Posner and Baecker [3] with some extensions and refinements. The 'work mode' component was added to describe the physical distance between collaborators and the synchronicity in 'time' of editing activities. Members of a collaborative editing group can work in the same office (same location) or in different locations (remotely). Editing activities can be in real-time (synchronous) or in different time slot (asynchronous).

Generally, many researches which were conducted in the 1990s and early of 2000s are based on interviews. They discovered different aspects of Collaborative Editing such as user roles, editing activities, writing strategies, and document control methods (Posner's Taxonomy of Collaborative editing) ; work-modes and time-synchronicity (Lowry's Time-Space Taxonomy of Collaborative editing). Some collaborative editing tools were developed that support 'synchronous' work-modes such as GROVE [7] - a multi-user outlining tool or ShrEdit [21] - a multi-user text editor. However, they are mostly used for research experimenting purpose.

Recent researches about 'Collaborative editing' start using the trace of collaborative editing tools such as Google Docs, Etherpad as an important corpus. Birnholtz et at [68, 30] parsed Google Docs revision histories and collected the editing logs to analyze the relationships between communication and editing in synchronous and asynchronous work modes. The authors reveal that edits and comments in CE often carry social meaning i.e. they can have emotional and relational impact [68]. They also pointed out that communication can be used to explain potentially conflicting behaviors and avoid negative relational effect. Their follow up research [30] presented an experimental study of group maintenance in collaborative editing using Google Docs. This is the first research according to the literature that consider to analyse editing logs. Editing logs were collected by paring the Google Docs revision histories. However, their study is strongly controlled by the authors as it was separated into two phases : asynchronous and synchronous. More over, the authors focused uniquely on the relationship between communication and editing as well the collaborator's social relationships.

Sun et al [28] analyzed the logs of editing activities using Google Suite of Google employees over two years to see how 'collaborative editing' has grown up at Google. They found that collaboration editing has grown rapidly up to 53% during the period they examined and 'concurrent editing is sticky', 76% of the employees who participate in a 'concurrent session' will do so again in the following month.

Wang et al [69] visualized the revision histories of Google Docs documents. They developed a tool, DocuViz, that can display who has contributed what and which changes were made to comment from them. This tool is 'potentially' useful to authors, instructors and researchers that are interested in collaborative editing 'patterns'.

Olson et al [29] examined the logs of collaborative editing behaviors using Google Docs of undergraduate students to see 'how people write together now'. They found that 95% of documents have some simultaneous work (i.e have at least one 'co-authored session'). The authors then focus on the quality of the document (i.e the output of collaborative editing process) and different aspects of Collaborative editing based on the 'Taxonomy of CE' [7, 4].

D'Angelo et al [31] presented a study of Space-time characterization of Collaborative Editing

based on Etherpad - an online real-time text editor. They classify edits into 'space' (i.e the position in a document) , 'time' and 'space-time collaborative edits or 'none' collaborative edits. They found that simultaneous editing happens very rarely within a predefined *'time-position window'*. They then focused on the proportion of 'space collaborative edits', 'time collaborative edit' and 'space-time collaborative edit' over the total number edits of the document. In another way, they consider the editing activities of whole document as a unique session (i.e one 'big' session for one document).

In fact, people are free to collaborate, they do not edit simultaneously on the whole document. In another way, the process of editing a document includes several editing sessions. Some sessions are edited by only a single author (called 'single authored session' and some sessions are edited by two or more authors (called 'co-authored session'). In their analysis, Sun et al [28] use a fix '15 minutes interval' to split a Google Docs document into several '15 minutes editing sessions'. These editing sessions are classified into 'co-authored session' or 'single authored session'. This approach has edge cases in which two collaborators edit the same document within 15 minutes period but their edits belong to two adjacent '15 minutes sessions'. They would not be counted as concurrent events. In a different approach, Olson et al [29] used a '7 minutes maximum gap' to group edits of a document into 'editing session'. It means that if two edits have 'time-distance' longer than 7 minutes, they belong to two adjacent editing session. Editing sessions then were classified into 'single-authored sessions' and 'co-authored sessions' based on the number of authors who have edits in those sessions.

Above are different approaches to determinate 'concurrent sessions' (i.e. 'co-authored sessions'). Although the document revision histories provided by Google Docs capture the details of edits at keystroke level, they were usually grouped into a 'slice' and assigned a single time-stamps [29]. It means that a series of insertions and/or deletions which are performed in a short period have the same time-stamp. This prevents a more fine-grained analysis of editing actions in such as short time period.

2.4 Awareness in collaborative editing

'Awareness' in general is the knowledge about a dynamic environment. This knowledge includes the state of some environment and also the interactions of people with the environment. As the dynamic environment changes over time, this knowledge should be maintained through perceptual information collected from the environment [70]. In collaborative work, the 'dynamic environment' is usually the 'shared work-space' among all group members. It could be a shared physical work-space such as an office with chalkboard or a shared virtual work-space such as a Git repository or an online ShareLaTeX project. The shared physical work-space allows people to maintain easily the 'knowledge' about others' activities and intentions (i.e future activities). In contrast, it's more difficult to maintain these knowledge in shared virtual work-spaces provided by collaborative systems (i.e groupware). Narrowing the scope of 'awareness' to collaborative work, Dourish et al [71] defined 'Awareness' as the understanding of activities of other individuals in the group. This definition is the most accepted and used widely in CSCW community.

According to Gutwin et al [2], CSCW researchers have proposed four types of awareness which are *Informal awareness, Group-Structural awareness, Social awareness and Work-space awareness*. These types of awareness overlap to some extent and interact to each other during collaboration process.

— *'Informal awareness' (also called 'presence awareness')* gives the general sense of who is active and what they are doing or are going to do. This information is what people know

when they work together in the same office.

— *'Group-Structural awareness'* brings information of group's organization such as people's roles/positions and their responsibilities. It also includes people's status and group's processes.

— *'Social awareness' (also called 'Conversational awareness')* is the information such as emotional state, level of interest that one person maintains about other collaborators during the conversation. 'Social awareness' is considered as an extension of 'Informal awareness'.

— *'Work-space awareness'* gives the 'up-to-the-minute' understanding of interactions of other collaborators with the shared work-space. It includes knowledge about where people are working, what changes they are making and also their future intentions. 'Work-space awareness' focuses on the interactions between people and the work-space rather than the work-space itself.

Among them, 'Work-space awareness' is widely used in general awareness literature [72]. Table 2.1 presents a set of key elements and their associated questions of the 'Work-space awareness'. These elements play an important role in many following researches in collaborative work, specially in distributed collaborative software development. Knowing of who in the group is working on what part of the coordinated project could help to minimize potential conflicts. It makes supporting for awareness become an important requirement in designing collaborative systems.

Element	*Relevant Questions*
Identity	Who is participating in the activity ?
Location	Where are they ?
Activity Level	Are they active in the Work-space ?
	How fast are they working ?
Actions	What are they doing ?
	What are their current activities and tasks ?
Intentions	What are they going to do ?
	Where are they going to be ?
Changes	What changes are they making ?
	Where are changes being made ?
Objects	What objects are they using ?
Extents	What can they see ?
Abilities	What can they do ?
Sphere of Influence	Where can they have effects ?
Expectations	What do they need me to do next ?

TABLE 2.1 – Elements of Work-space awareness. Source : Gutwin et al[2]

Dourish et al [71] discussed different approaches in providing a group awareness information in Collaborative Editing system. The authors then focus on the 'Shared-feedback' approach which automatically collects and distributes information of individual users' activities within the shared work-space. This approach can be applied for asynchronous, synchronous and also semi-synchronous work-spaces and systems.

Gutwin et al [73] in his study about how people manage a *'general awareness'* on three open-source software projects : *NetBSD, Apache httpd, Subversion*, found that official partitioning (i.e

each developer works in particular module or set of files) which reduces awareness requirement is limited. And developers were able to maintain a *'good general awareness'* of activities of other developers through text communication tools including *' mailing list', 'text chat' or 'commit log'*. By reading mailing list, commit log (commit summary/description) and 'overhearing' of informal and work-related discussion (text chat), people can gather important awareness information. They can also easily find out *'who are the experts in an area'* by sending a question to the group and then the 'right people' will join the conversation. However, gathering awareness information manually by reading text based artifacts is time-consuming task to new joining members.

Tam et al [74] developed a framework for asynchronous change awareness in collaborative documents and work-spaces. They figured out that '*who* changed the artifact', '*what* those changes involve, '*where* changes occur', '*when* changes were made', '*how* things have changed' and '*why* people make changes' are critical 'change awareness information' that a 'change awareness' system should provide.

Dewan et al [75] presented the semi-synchronous distributed computer-supported model that allows developers to detect and resolve potential conflicting tasks 'synchronously' while working 'asynchronously'. Rather than the traditional model in which conflict is detected during the 'check-in' (integration) phase, their mode provides the 'early conflict detection' which is based on the information of the dependencies among program elements such as files, classes, methods during the 'checked-out' phase. In another way, if a developer checkouts a file which is being edited by another developer, both of them will get a warning message. They may keep updating of changes from each other by setting up a 'watching flag'. And one of them may delay or adapt his/her editing activities to avoid conflict with the other one.

Ignat et al [76] proposed an awareness mechanism that helps developers to avoid blind modifications by localizing and re-presenting concurrent changes. This 'envisaged annotation' system provides to a user over their document the annotation of the source code with changes performed by other users. Users are therefore aware of concurrent changes without actually integrating remote changes. This awareness mechanism requires that users are connected to the network most of the time, even when they work in 'asynchronous' mode.

Brun et al [77, 15] presented an approach called 'speculative analysis' to help developers early detect and resolve conflicts in collaborative editing based on version control system. In fact, this approach does not 'speculate' potential conflicts but it performs 'merge' (of version control system), 'build' (with provided build script) and 'test' (with provided sets of test-cases) to detect 'textual conflict' and 'higher-order conflict (i.e 'build failed' or 'test failed').

In a similar effort, Kasi et al [16] conducted a 'retrospective study' in four open-source project on GitHub. They used the 'conflict minimization technique' that proactively identifies potential conflicts, encodes them as constrains and solves these constrains by recommending a set of conflict-minimal 'development path' for the development team.

According to the literature, most of previous studies about work-space awareness in collaborative editing focused on the systems that support 'asynchronous' collaborative work mode, especially version control systems. It can be explained by the fact that in 'synchronous collaborative work mode, users are aware of changes from other members. In another way, it is easier to avoid conflict in real-time collaborative editing as changes from one user are observed immediately by others. For instance, in real-time web-based collaborative editors such as Google Docs, Overleaf or Etherpad usually come with the multi-color cursors and each user is assigned a color cursor that is different from the others. When two users edit close to each other, they are intuitively aware of other's modifications. However, in some case they may make overlap edits that lead to conflicts afterward. This could be caused by the delay in real-time collaborative editors, especially in the case there are many users edit at the same time [78]. According to Ignat et

al, the delay in collaborative editing increases grammatical errors and redundancy (i.e increases textual conflict) [79].

Figure 2.23 illustrates an example of conflict in a real-time collaborative editor when two users edit the same word at the same time. In this example, two different users **'D'** and **'H'** edit very close to each other in both time and position (covered by the red ellipse). User **'D'** makes a typo, he inserts *'prdefined'* instead of *'predefined'*. User **'H'** tries to help, he inserts *'e'* between *'r'* and *'d'*. Meanwhile, user **'D'** also comeback to fix his typo. He inserts another *'e'* between *'r'* and *'d'*. The final result is *'preedefined'* instead of *'predefined'*. And it needs one of them to correct again the final result (by removing one character *'e'*).

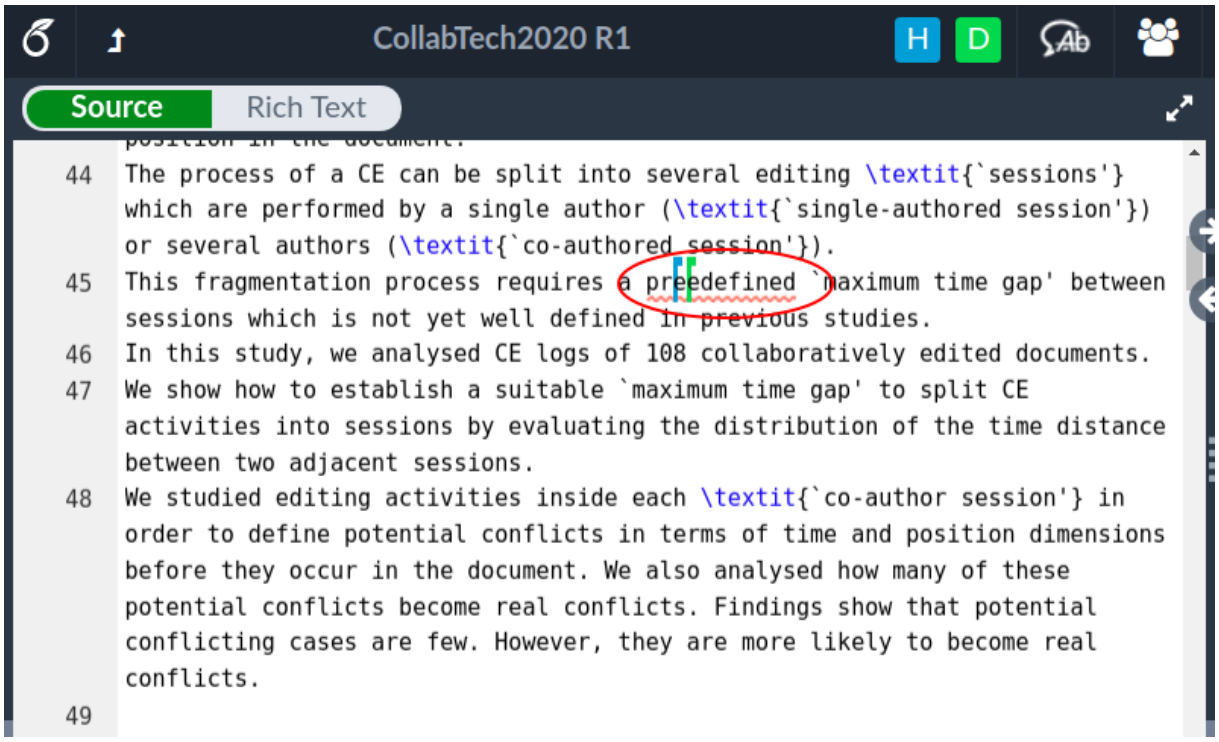


FIGURE 2.23 – An example of conflict in a Real-time collaborative editor

We are interested in how to support 'work-space awareness' in real-time web-based collaborative editor such as ShareLaTeX, especially the cases in which two users edit very close to each other and they have high potential to make overlap changes.

2.5 Chapter conclusion

In this chapter, we have presented the basic notions of collaborative editing including version control systems and real-time web-based collaborative editors. We focus on the detail working mechanism of Git, a decentralized version control system and also the synchronisation algorithms used in real-time collaborative editor. We then introduce some previous studies of collaborative editing based on different tools and used different approaches. Finally, we introduce about 'awareness' in collaborative editing and some previous work related to 'work-space awareness'.

The lack of a detailed analysis of conflict in decentralized version control system such as Git gives us motivation to conducted a study of conflict and resolution in Git-based open source projects (Chapter 3). And the lack of a study of editing activities based on real-time web-based

collaborative editors gives us motivation to conducted a study of time-position characterisation of conflict in collaborative editing using ShareLaTeX. (Chapter 4).

Chapter 3

Merge Conflicts and Resolutions in Git-based Open Source Projects

3.1 Introduction

In this chapter, we present our study about merge conflict and resolution in Git-based open source projects.

Version control systems make it easy for users to work in parallel on a shared project. It means that the project is easy to be diverted, i.e two users have two different version of a project and they need to 'synchronize'/'integrate' these two versions to have a common result. Version control systems support the 'merge' function to merge parallel changes made by two different users. Allowing concurrent changes is very important to support collaborative work, however, merging different parallel changes might require a significant period of time which can alter the productivity gain. The most basic and effective merging technique is 'textual merging' [20]. 'Textual merging' consider software artifacts (i.e text files, source files, configuration files) as flat text files and the indivisible unit (i.e atom) are the lines of text. This is also called as line-based merging approach.

Studies showed that in large projects the partition of software modules among developers is limited and developers can contribute to any part of the code [73]. It means that two users can edit the same file concurrently. If they edit in different part of the file, their changes are 'conflicting' and need to be integrated. Most of version control systems can merge concurrent changes of the same file automatically. However, if they edit at the same part of the file which is usually the same line or two adjacent lines, the system can not merge their changes successfully. This case is denoted as 'unresolved conflict' and the users has to resolve it manually. Unresolved conflicts also occur if a file is renamed and modified/deleted concurrently, if it is modified and deleted concurrently, if it is renamed concurrently by two users, if a user renames a file with the same name as another user gives to a concurrently created file and if two users concurrently add two files with the same name. Note that by the name of a file we understand the whole path identifying that file.

Conflicts are costly as they delay the development process [67]. In the period of time between conflicts occur and they are discovered and understood, they might grow and become difficult to resolve. Developers may postpone integrating parallel work as they fear that conflicts may be hard to resolve. This concern of potential conflicts makes parallel work to diverge more and conflicts are more likely to happen and grow. In this work, we studied the conflict of concurrent editing activities in Git, the most popular decentralized version control system [17]. In Git, users

can synchronize their changes with other users working in parallel with them. In this process, a merge is performed between local changes and remote changes. Conflicts could happen during this merging process. Understanding how often and when conflicts are more likely to happen during the development process and how users resolve them can help proposing awareness mechanisms that can prevent conflicts from happening. This study could help proposing better merging approaches that minimize conflicts that users have to manually resolve. We analyzed traces of projects developed with Git in order to quantitatively analyse the different types of textual conflicts at the level of files that arise at the different development cycle phases. One particular type of unresolved conflict that we study is that referring to adjacent lines. If concurrent changes occur on two adjacent lines Git signals an unresolved conflict, but not in the case of two lines separated by two or more lines. As there is no reason why these cases are treated differently, we aim studying whether developers resolve them differently. We also aim quantitatively measuring merge user satisfaction after a conflict resolution in terms of how often users roll back to a previous version. Even if several existing studies focused on parallel changes and conflicts on Git-based projects [15, 16], they did not analyse fine-grained conflicts at file level and their resolution mechanisms.

Several tools rather than using textual merging, use syntactic or semantic merging [20]. Syntactic merging takes the syntax of software artifacts into account, while semantic merging considers semantic information. Studies such as [15] and [16] considered both textual conflicts as result of textual merging and higher order conflicts that are conflicts at semantic level that cause compilation errors or test failures. Other studies such as [49] studied indirect conflicts when changes to one software artefact affect concurrent changes to another artefact. In [49] authors proposed the social call graph that describes dependencies between software developers for a piece of code. The social call graph combines the call graph data structure that contains all the dependency relationships of a software application with authorship information. Our study does not investigate indirect and higher level conflicts and focuses uniquely on conflicts related to the same file with a particular attention for textual conflicts as used by main DVCSs such as Git.

The rest of this chapter is organized as follows. Section 3.2 presents our conflicts measurement during the merge process. Section 3.3 discusses implications for design for our study and some limitations of analysing Git repositories. Section 3.4 gives some concluding remarks.

3.2 Measurements

In order to measure the level of parallelism and the proportion of conflicting modifications in DVCSs, we adopted an experimental methodology where we analysed the corpus of four large open-source projects developed using Git :

— **Ruby on Rails** [32] is a web framework, with integrated support for unit, functional, and integration testing. We analysed version 5.0.0.alpha of this project.
— **IkiWiki** [33] is a wiki software system that compiles wiki pages into HTML pages for publication. We analysed IkiWiki version 3.0.
— **Samba** [34] is an implementation of networking protocols to share files and printers between Unix computers and Windows computers. We analysed Samba 3.0.x.
— **Linux Kernel** [35] is an implementation of a Unix-like computer operating system kernel. We analysed version 4.x of the Linux kernel.

Beside the large size and the popularity of these projects, they are representative for the different software development pull-based [46] models that they adopt. In practice, the core-development-team will organize at least one repository as the primary repository where the latest

approved changes can be found. Contributors can clone from this official repository. However, only the core-development-team has the write-access to commit directly. Other contributors need to use the pull-based development model in which a contributor creates a pull-request for his changes. A core-team's member then inspects the changes and decides to pull and merge contributor's changes to the main repository or not. And in some cases, contributors are requested to update or add more changes before their pull-request is accepted. Nowadays, the pull-based model is naturally supported by web-based hosting services such as [47] and [48].

Rails project uses pull-request model which is naturally supported by [47]. Contributors can fork (clone) from the official GitHub repository and contribute via GitHub's pull-requests. In a pull-request, reviewers and its contributor communicate directly using pull-request's comments. These comments are available to other users and they can participate into this conversation. Afterwards a pull-request can be merged to the main line or declined.

IkiWiki looks like a *private repository* where contributors send their *patches* to Joey Hess, the main developer of the project.

Samba uses a *shared repository* among registered contributors. It uses an auto-build system for code-review process. Contributors need to join a technical mailing-list before contributing.

Linux Kernel uses a pull-based model via mailing-list. Contributors need to send their *patches* to the appropriate subsystem maintainer's mailing-list in charge of the different parts of the project.

Table 3.2 presents some details about these projects : the period of their development (until 05-October-2015), the number of commits, the number of contributors (authors), the number of created files during the lifetime of the project and the number of existing files on 05-October-2015. Note that if a file is moved during the lifetime of the project from a place to another, we counted it as a new created file.

Project name	*Period (days)*	*No. of commits*	*No. of authors*	*No. of created files*	*No. of existing files*
Rails	3,967	53,625	3,422	10,272	2,984
IkiWiki	3,496	19,375	982	4,610	3,362
Samba	7,094	100,301	386	33,626	7,582
Kernel	5,132	547,515	14,395	90,173	51,567

Table 3.1 – Open source projects developed using Git

In contrast to CVCSs, Git does not support the centralized logging feature of all user activities. The best overview of user activities is provided by the commit history (including merges) from the primary repository. To identify concurrences and conflicts in each project, we created a *shadow* repository and recursively re-integrated developer's changes into this repository. In other words, by means of Python scripts [80] we re-played all merges that were performed during the development period of each project.

3.2.1 Integrations and conflicts on files

We first determined the number of concurrent updates to a same file and then the number of concurrent updates to a same file that resulted in unresolved conflicts. Similar to [14] we computed the *integration rate* and *conflict rate* as provided in Table 3.2. *File updates* represents the total number of updates to files. A file can be updated several times throughout the development cycle. *Integration rate* represents the proportion of concurrent updates to a same file over all

updates to files. *Conflict rate* is calculated by the proportion of updates to a same file that resulted in unresolved conflicts over concurrent updates to files. The file updates were collected from all commits of the project. And by re-integrating all developer's changes, we computed the concurrent updates to a same file and the concurrent updates to a same file that resulted in unresolved conflicts.

Project name	*File updates*	*Integration rate*	*Unresolved conflict rate*
Rails	117,960	4.04%	16.26%
IkiWiki	37,327	1.08%	50.50%
Samba	306,182	0.68%	87.84%
Kernel	1,278,247	10.99%	4.86%

Table 3.2 – Concurrency and conflicts on files

We can notice that Kernel and Rails projects have larger integration rate than IkiWiki and Samba. For instance, integration rate in Kernel project is 10 times larger than IkiWiki and 16 times larger than Samba. This can be explained by the large size of Kernel project in terms of the number of files. In contrast with the *integration rate*, Rails and Kernel have smaller *conflict rates* than IkiWiki (50.50%) and Samba (87.84%). We do know that Rails is a large project using advantages of GitHub, which supports pull-based model naturally. GitHub interface allows not only the author of a pull-request and the reviewer but also other contributors and core-team members to discuss about that pull-request and its issues. It brings a big advantage of sharing collaborators knowledge to solve problems during integration. In case of Linux Kernel, it uses pull-based model via mailing list with a list of subsystem maintainers. It also has a list of delegated servers, such as *linux-next*, where commits are tested before they are pushed to primary repository [61]. On the other side, Samba uses shared repositories among contributors and IkiWiki is maintained as a private repository by Joey Hess. Nowadays, all of them provide a list of Todo tasks and a list of Bugs where contributors can focus their work to avoid conflicting integration.

The lack of a central server that holds a reference copy of the project introduces more parallelism between user versions allowing them to diverge more in DVCSs than in CVCSs. For instance, Kernel, Rails, IkiWiki and Samba projects developed in Git have significantly (99% confidence level) higher *integration rate* (22, 8, 2 and 1.5 times respectively) than projects in CVS analysed in [14]. However, the higher *integration rate* does not result into higher *conflict rate*. For instance, Kernel and Rails have 5 and 1.5 times respectively lower *conflict rate* than projects in CVS whereas Samba and IkiWiki have almost 2 times higher *conflict rate*. The *conflict rate* in Git's projects depends on collaboration process management.

Project name	*Content* conflict	*Remove/Update conflict*	*Naming conflict*
Rails	89.68%	2.97%	7.35%
IkiWiki	46.31%	1.48%	52.22%
Samba	64.47%	34.44%	1.09%
Kernel	90.96%	8.63%	0.41%

Table 3.3 – Proportion of conflict types

We also measured the proportion of the different conflicts types : content conflicts referring to conflicts inside a file, remove/update conflicts referring to concurrent removal and update of a file and naming conflicts referring to concurrent renaming of the same file or of two files with the same name. Table 3.3 presents the proportion of conflict types of four projects. We found that content conflict is far the most popular type of conflict with a proportion of 46% - 90% from all conflict types.

3.2.2 Integrations and conflicts based on release dates

In the previous section we presented the *integration rate* and *conflict rate* of four projects over their whole development period. However during their development life-cycle, activities are not equally distributed. Our hypothesis is that collaborative activities achieve some peaks around project release dates, such as periods of one or two weeks before a release date. To gain a better understanding about collaboration during those active periods, we conducted an analysis about integrations and conflicts based on project release dates.

Figure 3.1 illustrates four active periods of one week length before and after respectively the release date (RD). We denote these periods as follows : B2W (between two weeks before RD and one week before RD), B1W (between one week before RD to RD), A1W (between RD to one week after RD) and A2W (one week after RD to two weeks after RD). We also analysed B4W, B3W, A3W and A4W.

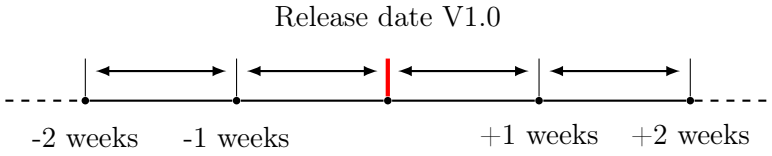


FIGURE 3.1 – Analysis based on release date

IkiWiki

There are three official versions of IkiWiki, however the first two versions did not introduce any concurrency nor conflict. In fact, at that time, IkiWiki was developed by Joey Hess only. We analysed the latest version V3.0 with a release date on 31-12-2008. The result is presented in Figure 3.2.

The result shows that one week before RD, the *integration rate* is very high (29.41%) with a conflict rate of 46.67%. Also, the *integration rate* decreases in the next two weeks after RD(A1W, A2W) and increases in the third and the fourth weeks after RD (A3W, A4W). All integrations in A1W, A2W and A4W are successful and all integrations in A3W generated conflicts.

This behaviour can be explained by the policy adopted by IkiWiki where contributions are merged by Joey Hess just one week before RD. Figure 3.3, extracted from IkiWiki official site [33], is an illustration for this explanation. Many merges and commits are submitted close to RD. There are still some minor integrations such as bug-fixing after RD. We also analysed the following four weeks after A4W (A5W to A8W) and did not find any integration nor conflict. So we can say that for IkiWiki, the integration of version V3.0 begins one week before the official release date and lasts four weeks after that release date.

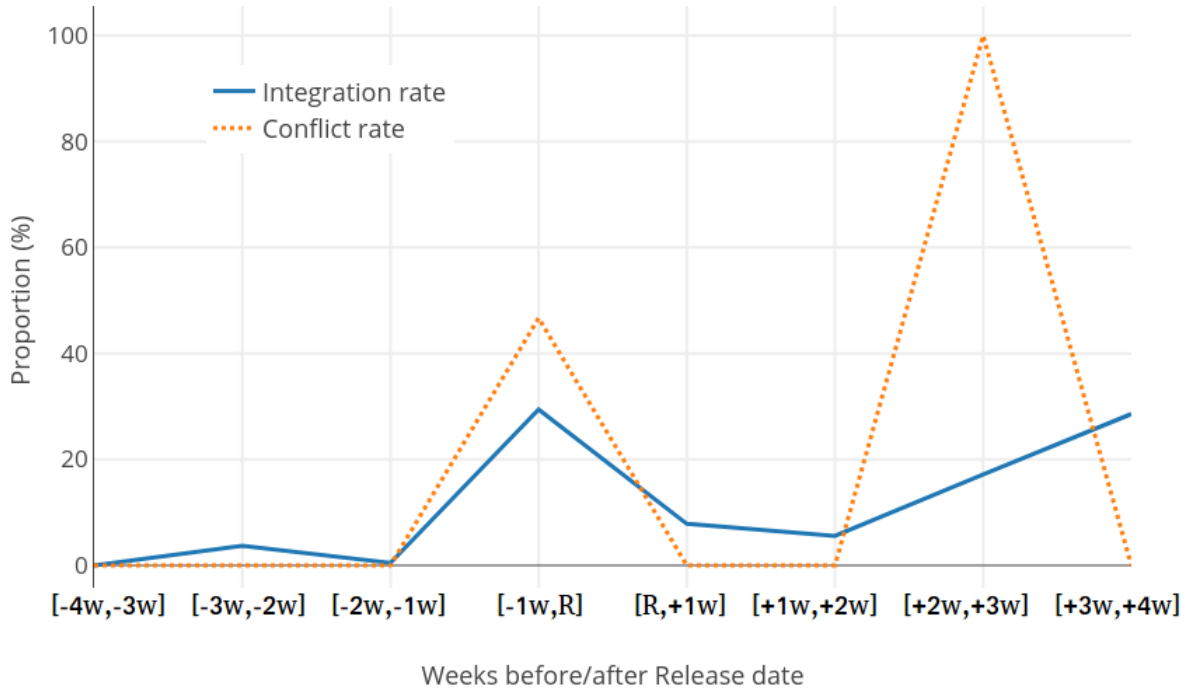


FIGURE 3.2 – IkiWiki-V3.0, integration and conflict rate on files

Samba

Samba community started using Git as its main repository from version 3.2. We chose to analyse three Samba versions : 3.2, 3.3 and 3.6 based on the number of merges between B4W and A4W. In fact we could not find any merges in the period (B4W-A4W) in other versions (4.0, 4.1, 4.2, 4.3). The results are presented in Figure 3.4 and Figure 3.5.

Figure 3.4 shows that the *integration rates* of version 3.2 and 3.3 are almost similar. They have high *integration rates* in the week before RD, in the second week after RD and in the fourth week after RD. These *integration rates* are 2 times (for V3.2, B1W), 6 times (for V3.2, A2W), 3 times (for V3.3, B1W) and 4 times (for V3.3, A2W) respectively higher than the overall *integration rate* (0.68%, Table 3.2).

The integration process in version 3.6 changed compared to versions 3.2 and 3.3. Version 3.6 had integrations only in the two weeks before RD and the first week after RD (B2W and A1W). Furthermore, its *integration rate* is three to six times lower than in versions 3.2 and 3.3.

In Figure 3.5, we can see that some integrations in version 3.2 and 3.3 in A2W period resulted in conflicts and all other integrations in B4W-A4W period are successful. Specially version 3.6 does not introduce a single conflict during B4W-A4W. In fact, starting from version 3.6, each release version officially has a series of pre-release (pre) versions and release-candidate (rc) versions. For version 3.6, it has three pre versions (3.6pre1, 3.6pre2, 3.6pre3) and three rc versions (3.6rc1, 3.6rc2, 3.6rc3). It means that most collaboration works are done in pre and rc versions, not in official release version 3.6. It also explains why we could not find any merges in

2008-12-31	Joey Hess	formatting
2008-12-31	Joey Hess	fix link
2008-12-31	Joey Hess	Merge branch 'master' of ssh://git.ikiwiki.info/srv...
2008-12-31	Joey Hess	Merge branch 'next'
2008-12-30	intrigeri	Merge commit 'upstream/master' into prv/po
2008-12-29	Joey Hess	update
2008-12-29	Joey Hess	Merge branch 'master' into next
2008-12-28	Joey Hess	Merge branch 'master' into next
2008-12-26	Joey Hess	Merge branch 'master' into next
2008-12-26	Joey Hess	Merge branch 'master' into next
2008-12-25	Joey Hess	Merge branch 'master' into next
2008-12-25	Joey Hess	more 3.0 docs, changelog
2008-12-24	Joey Hess	typo

FIGURE 3.3 – IkiWiki-V3.0, commits in the week before RD

B4W-A4W of other versions after 3.6.

Rails

The first official version (1.0) of Rails was released in 13-December-2005. However, the most active periods started from version 3.1. We chose to analyse the following Rails versions : 3.1, 3.2, 4.0, 4.1 and 4.2. The results are presented in Figure 3.6 and Figure 3.7.

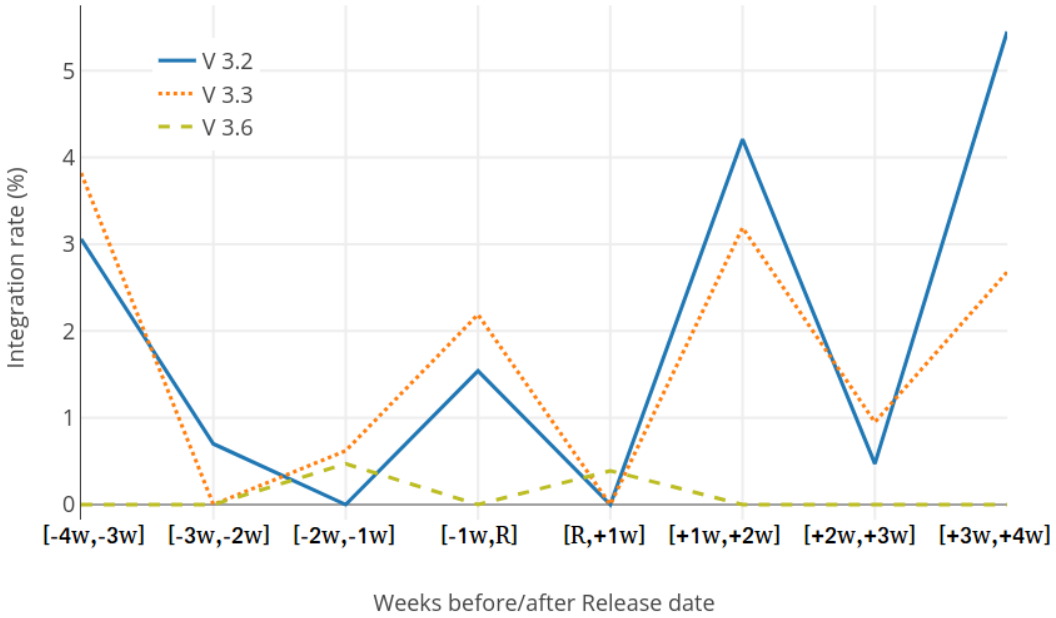


FIGURE 3.4 – Samba, integration rate on files

Versions 3.1 and 4.2 have high *integration rates* in B3W-A3W periods. Version 3.2 and 4.0 have two distinct active periods of integration : B3W-B2W and A1W-A3W. Version 4.1 has three distinct active periods of integration : B4W-B3W, B1W-A2W and A4W. Generally, for versions 3.2 and 4.0 the integration was done in the two weeks before the release date (B3W-B2W). Their *integration rates* in B1W are very low : 0.83% (V3.2) and 1.26%(V4.). However, only version 3.2 is conflict-free in B1W while version 4.0 has a high *conflict rate* (30%). In contrast, versions 4.1 and 4.2 have high *integration rates* in B1W : 30.93% and 18.48% respectively for V4.1 and V4.2. They also have high *conflict rates* in B1W : 30.93% (V4.1), 23.08%(V4.2). We can see that the integration process in Rails had changed from version 3.1 with a high *integration rate* in B1W to versions 3.2 and 4.0 with a very low *integration rate* in B1W. Then the *integration rate* became very high in versions 4.1 and 4.2.

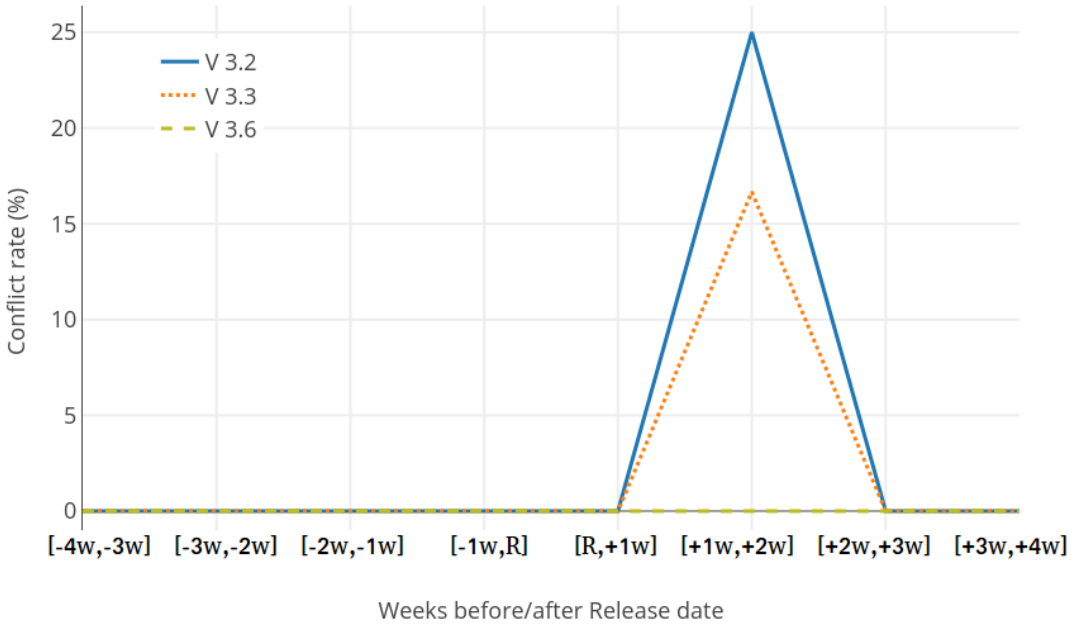


FIGURE 3.5 – Samba, conflict rate on files

Linux Kernel

Linux Kernel community started using Git as the main repository in version 2.6.x. We chose to analyse the versions 2.6.39, 3.0, 3.1, 3.19, 4.0 and 4.1. Each version was developed in 2-3 months. The versions 2.6.39, 3.0, 3.1 and 3.19 were released in 2011 while 4.0 and 4.1 were released in 2015. We analysed how the collaboration process in Linux Kernel was changed over years.

Figure 3.8 shows that with the exception of V3.19, the five other versions have almost similar integration trends with an active period of integration of three weeks after RD(A1W-A3W). It has 4-8 times higher *integration rate* than other periods. For version 3.19, the active period of integration is two weeks before RD(B2W-B1W) with a 7 times higher *integration rate*.

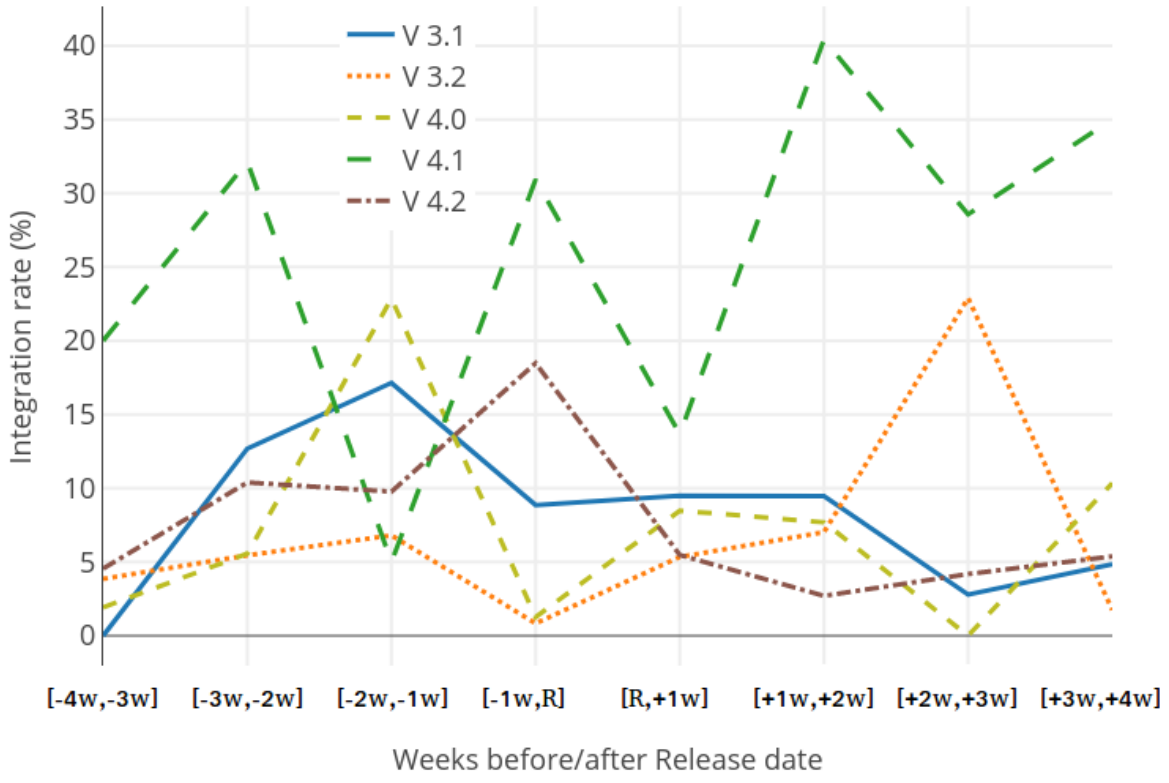


FIGURE 3.6 – Rails, integration rate on files

In fact, Linux Kernel community uses a development process called 'merge windows'. At the beginning of a new version development cycle - right after the official release date of previous version - the 'merge window' is opened. The bulk of changes is merged during this time. The 'merge window' lasts for approximately two weeks. After this period a series of release-candidates (rc) are proposed over the next six to ten weeks. On this time, only patches which fix problems are allowed to be submitted to the mainline. This explains why in Linux Kernel, the most active period of integration is A1W-A3W. Note that all the changes integrated during the merge window have been collected, tested and staged ahead of time. It keeps the *conflict rate* of Linux Kernel in this period quite low (3.6%-8.39%) in comparison to active periods of other projects such as Rails (40%, A2W), Samba (25%, A2W), IkiWiki (46%, B1W). Additionally, Figure 3.9 shows that *conflict rate* in B2W is sightly higher than in A2W although A2W is the most active period of integration. We can explain that integration before the RD has higher chance to result into conflicts.

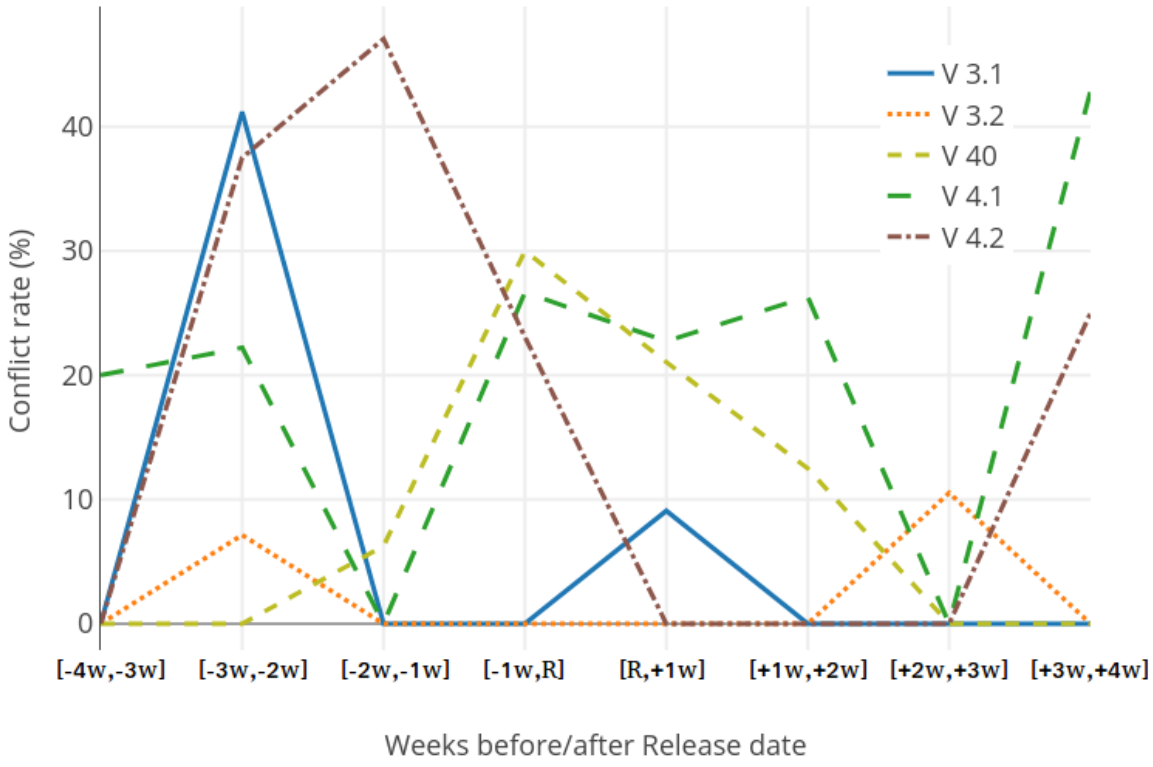


FIGURE 3.7 – Rails, conflict rate on files

Overall

By analysing the collaboration process of these projects at specific time periods which are close to release dates, we can reveal how change integration evolves over time. In IkiWiki (3.0), Samba (3.2, 3.3) and Rails (3.1), an 'old integration style' was found in which most integration works were submitted just one week before release date (B1W). This style was changed in the next versions of Samba (3.6) and Rails (3.2, 4.0) that is integration works were submitted two weeks before release date. Moreover, the integration process of Rails was changed also in its next versions (4.1, 4.2). This time, it became worst with very high *integration rate* in B1W (2.5-3.5 time higher than in version 3.1). Linux Kernel has a more stable integration process which is called 'merge window'. This integration process was changed sightly over its versions. With this analysis, we can know when Samba community started using pre-release and release-candidate versions and how the 'merge window' is used effectively in Linux Kernel. However, not all the stories are revealed such as why version 3.19 has a different active period (B2W) in comparison with other Linux Kernel versions.

Table 3.4 presents the proportion of conflict types based on Release date. Our analysis revealed that content conflict *'CONFLICT (content)'* is the most popular conflict type with 93.3%, 100%, 76%-100%, 81.85%-97.41% over all conflicts respectively in IkiWiki, Samba, Rails and Linux Kernel.

In addition, we measured the Spearman's rank correlation coefficient (Rho) [81] between integration rate and conflict rate. We collected data points based on the release date. The result which is presented in Table 3.5 shows that the correlation between them is none or very weak in

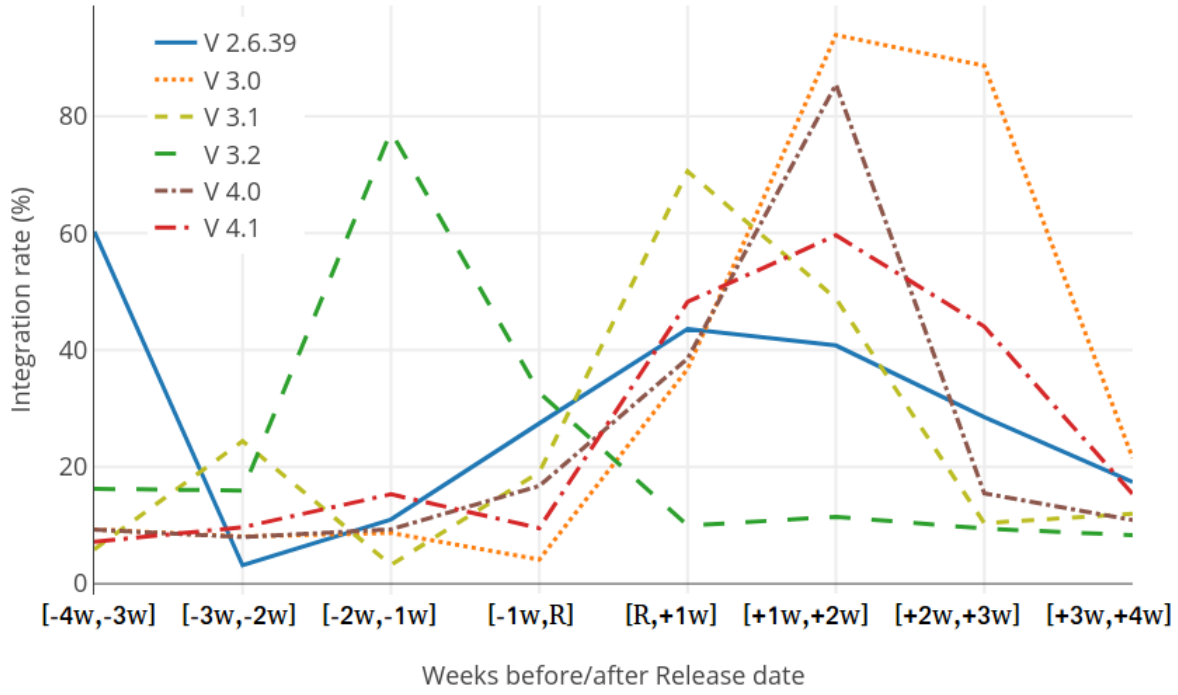


FIGURE 3.8 – Linux Kernel, integration rate on files

both Linux (48 data points, negative monotonic) and Rails (18 data points, positive monotonic). We do not have enough valid data points for IkiWiki and Samba.

3.2.3 Conflict resolution

Several files can be in conflict during a merge. We counted the total number of merges performed during the lifetime of the project and the number of merges resulted in unresolved conflicts. Table 3.6 gathers these results of all projects analysed. When a merge is not resolved automatically by Git, users need to resolve it manually. A user can decide to rollback to a previous version. We provide also the rollback rate, i.e the number of times user uses rollback action over all the merges that contain unresolved conflicts. The rollback rate is lower in Kernel and Rails compared to IkiWiki and Samba.

In practice, after a successful merge, users build and test the merging results. A successful merge can result in build-failed or test-failed. In order to be considered as a *successful integration*, a successful textual merge needs to be built and tested successfully also. Otherwise, it is considered as *higher-order* conflict [77]. To detect these conflicts, we have to build using provided project build-scripts and test using provided test-sets every successful merge. We did not measure these types of conflicts as not all projects provided test-sets.

We provided in Table 3.7 a more detailed analysis of the rollback action in which we find rollback actions in all merges, not only the ones that contain textual-conflicts. *Level* represents the number of following commits after a merge when the rollback actions happen. Figure 3.10 illustrates *levels* of rollback in which branch *B2* is merged to branch *B1*. The merge result is *M0* and *C1, C2, C3* are the following commits after the merge.

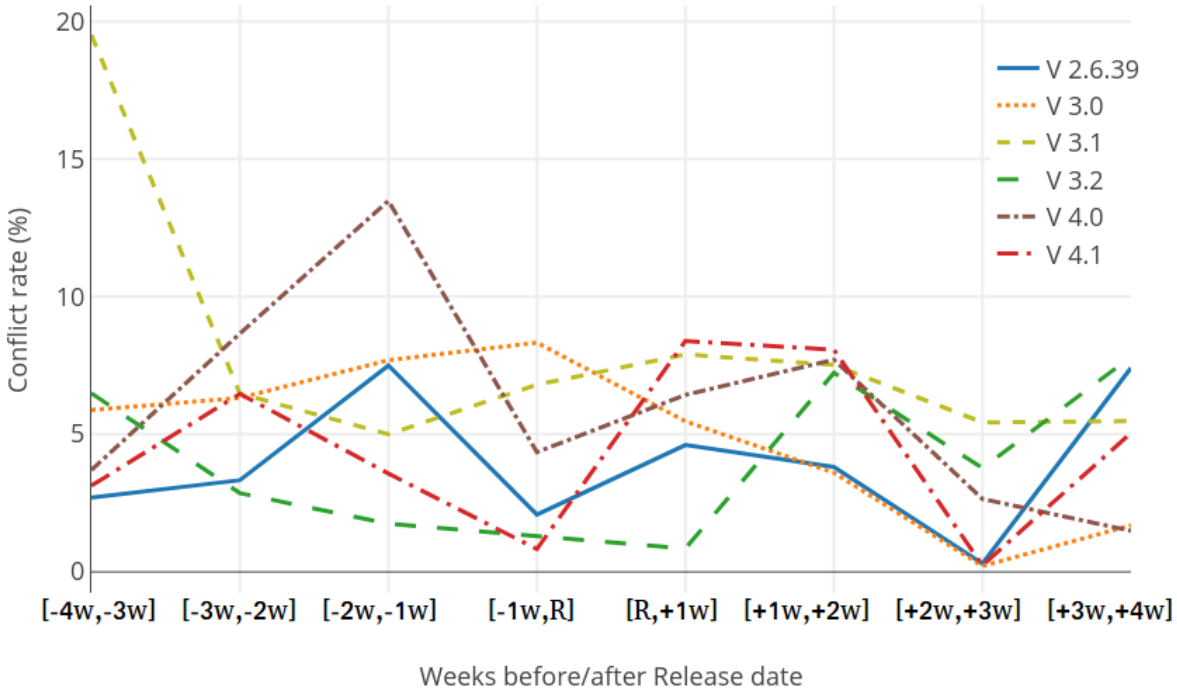


FIGURE 3.9 – Linux Kernel, conflict rate on files

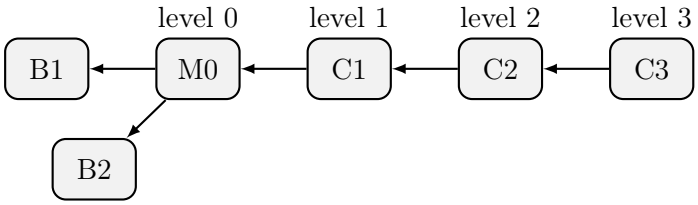


FIGURE 3.10 – Levels of rollback action

Normally, if a user decides to rollback, he can use the *'git revert'* command to rollback the merging. The *'git revert SHA-1-B1'* usually creates a new commit after the merging commit (i.e commit *C1* has the same *SHA-1 hash* with *B1*). We assigned *level=1* to this case. However, in the most of the cases, users don't want to create new commits. Users can use a set of commands to rollback without creating a new commit such as *'git checkout SHA-1-B1 ; git reset –soft HEAD ; git commit -amend'* that will rollback *M0* to *B1* version (i.e commit *M0* has the same *SHA-1 hash* with *B1*). We assigned *level=0* to this case. Note that, in practice, when merging a contributor's work to the main repository, the core-team members can use *'git merge -s ours'* to chose the main-line version as the default result when conflicts happen or use *'git merge –no-commit'* to test the merging results and manually fix the conflicts before committing them. Furthermore, the rollback actions can happen in the next one, two or three commits. We assigned *level=1, level=2, level=3* to these cases. In our measurements, we limit *level* to four. In fact, we could not find any rollback actions after three commits from the merging result.

Table 3.7 shows that Rails (33.46%) and Samba (20.45%) have many more rollback actions than IkiWiki(5.21%) and Kernel(5.18%).

Project name	*Content* conflict	*Remove/Update conflict*	*Naming conflict*
Rails	89.68%	2.97%	7.35%
IkiWiki	46.31%	1.48%	52.22%
Samba	64.47%	34.44%	1.09%
Kernel	90.96%	8.63%	0.41%
IkiWiki 3.0	93.33%	6.67%	0.005%
Samba 3.2	100.00%	0.00%	0.005%
Samba 3.3	100.00%	0.00%	0.005%
Samba 3.6	0.00%	0.00%	0.005%
Rails 3.1	62.50%	0.00%	37.505%
Rails 3.2	100.00%	0.00%	0.005%
Rails 4.0	100.00%	0.00%	0.005%
Rails 4.1	100.00%	0.00%	0.005%
Rails 4.2	76.00%	4.00%	20.005%
Linux 2.6.39	96.12%	3.88%	0.005%
Linux 3.0	97.41%	2.59%	0.005%
Linux 3.1	81.85%	17.74%	0.005%
Linux 3.19	97.26%	2.74%	0.005%
Linux 4.0	95.16%	4.84%	0.005%
Linux 4.1	94.49%	4.72%	0.005%

TABLE 3.4 – Conflict types based on release date

Project name	*No. of* data points	*Rho*	*P-value*
Rails	18	0.102	0.343
IkiWiki	2	NA	NA
Samba	2	NA	NA
Kernel	48	-0.188	0.1

TABLE 3.5 – Spearman's rank correlation coefficient of integration rate and conflict rate

3.2.4 Adjacent-line conflicts

In this section we analyse Git mechanism of considering concurrent modifications of two continuous lines as being in conflict (called adjacent-line conflicts). Figure 3.11 illustrates an example of adjacent line conflict with both expected and real merge result. User at site-1 makes changes on line 2 and user at site-2 makes changes on line 3. They then merge their changes and Git generates a *'CONFLICT (content)'*. Our hypothesis is that this is not a content conflict because these changes are made on two different lines. Git should merge them successfully by applying changes from both sites. To test our hypothesis, we analysed all content conflicts in the four projects to detect all adjacent-line conflicts. We then analysed the adjacent-line conflict resolutions that were manually fixed by the authors to check if both changes done on the adjacent lines were applied. If in most cases both changes were applied we can conclude that adjacent-line conflicts should not be considered conflicts.

We used *'git difftool'* to detect the changed lines of two sites in comparing to the original

Project name	*No. of merges*	*Unresolved conflict merge rate*	*Rollback rate*
Rails	9,728	4.34%	1.66%
IkiWiki	1,037	7.52%	5.13%
Samba	1,281	10.07%	6.98%
Kernel	38,961	9.11%	0.70%

TABLE 3.6 – Frequencies of conflicting merges

Project name	*Level 0*	*Level 1*	*Level 2*	*Level 3*	*Level 4*
Rails	3,217	36	2	0	0
IkiWiki	39	13	1	1	0
Samba	260	2	0	0	0
Kernel	2,016	1	0	0	0

TABLE 3.7 – Rollback action after merging

document. Figure 3.12 illustrates how the changed-lines is detected by denoting a changed line as 'X' and an unchanged line as 'V'. After updating the changed-lines for each *content conflict*, we used adjacent-line patterns for each two continuous lines. Figure 3.13 (Group 1) illustrates adjacent-line conflicts where the first line is changed by only one site and the second line by only the other site.

Figure 3.13 (Group 2) presents four patterns including both an adjacent-line conflict and a normal content conflict. In our analysis of adjacent-line conflicts, we excluded these patterns by considering that they can be resolved as normal content conflicts. In our analysis we considered only adjacent-line conflicts which do not include normal content conflicts.

The results are presented in Table 3.8 illustrating the number of adjacent-line conflicts and their resolutions. Three types of resolutions exist : applying both changes, applying a change from one site only (either from site-1 or site-2) and applying no changes.

The proportions of applying both site changes are 24.39% in Samba, 57.5% in Rails, 75% in IkiWiki and respectively 85.01% in Linux Kernel. We did the same analysis on how users resolve normal content conflicts and Table 3.9 presents a comparison of the results obtained for adjacent-line conflicts and normal-content conflicts.

We compared the frequency of applying both changes for adjacent-line conflicts to that of applying both changes for normal content conflicts. Table 3.10 presents for each project the

Project name	*Adjacent -line conflicts*	*Applying both changes*	*Applying one change*	*Other cases*
Rails	80	46	28	6
IkiWiki	4	3	1	0
Samba	41	10	23	8
Kernel	367	312	51	4

TABLE 3.8 – Adjacent line conflicts and resolutions

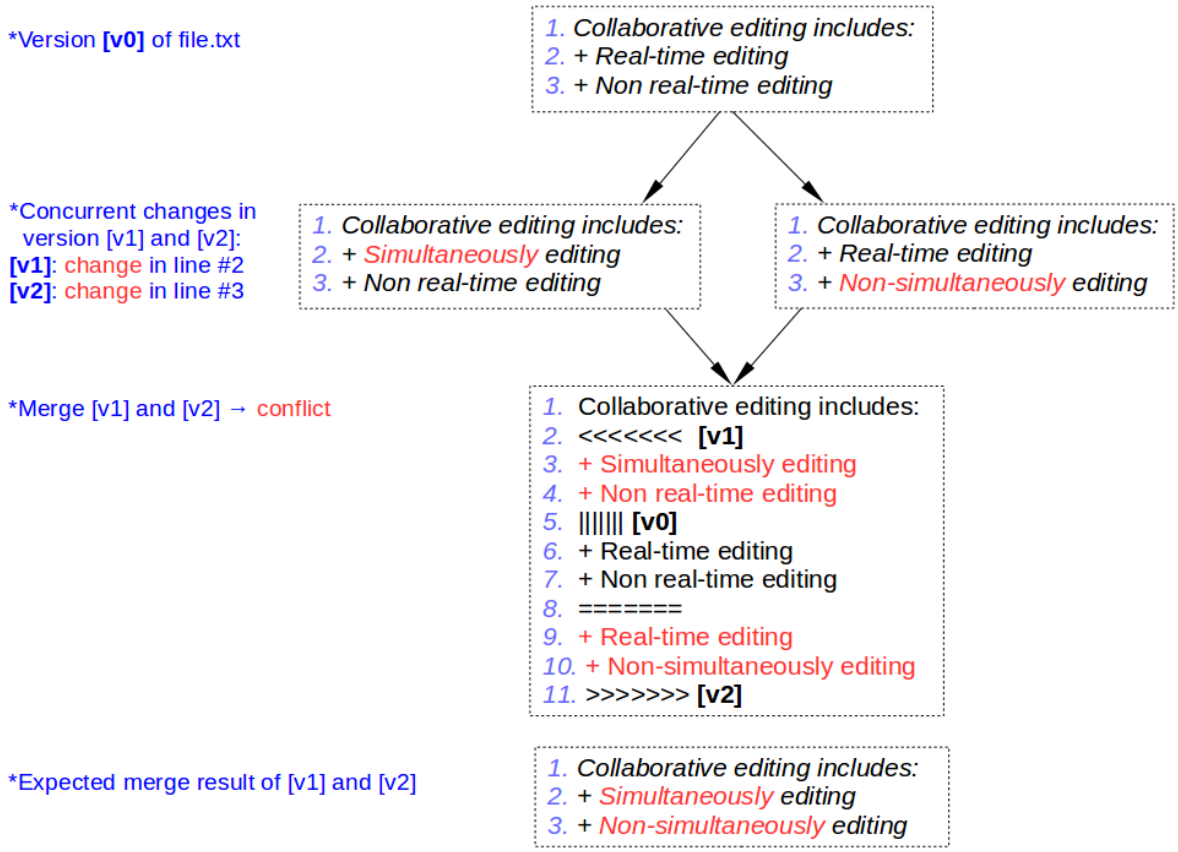


FIGURE 3.11 – Adjacent-line conflict : expected and real merge results

standardized mean-difference effect size (SdMD) between proportions of applying both changes for adjacent-line and normal content conflicts [82]. Only Samba has a low SdMD and its lower bound of confidence interval is less than zero. Excepting Samba, we obtained significant confidence at level of 95% that 'adjacent-line conflicts' are resolved more often by applying both changes than 'normal content conflict'. Applying both changes in the case of a conflict means that the concurrent changes were not in conflict, so the conflict was not necessary to be detected.

We had conducted an empirical test on adjacent-lines changes for Darcs and SVN [80]. The results show that SVN (CVCS) and Darcs (DVCS) merge changes in adjacent lines successfully.

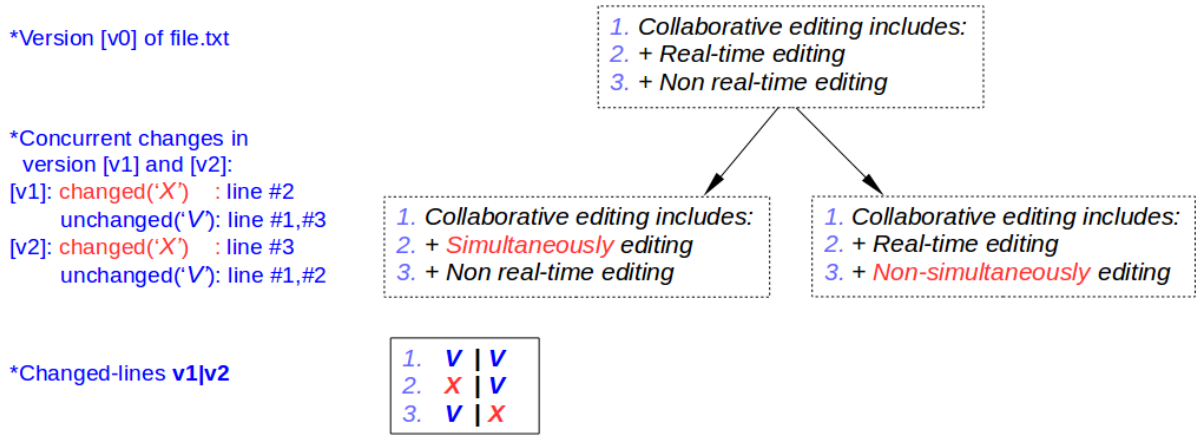


FIGURE 3.12 – Changed-lines

	Adjacent-line conflicts		Normal content conflicts	
Project name	*No. of conflicts*	*Applying both changes*	*No. of conflicts*	*Applying both changes*
Rails	80	57.50%	317	5.67%
IkiWiki	4	75.00%	22	9.09%
Samba	41	24.39%	1149	14.19%
Kernel	367	85.01%	1326	13.38%

TABLE 3.9 – Adjacent-line and normal content conflicts

3.3 Discussion

We found that content conflict is far the most popular type of conflict with a proportion of 46% - 90% from all conflict types. Compared to centralised version control systems such as CVS, in Git we have a significantly higher integration rate, i.e. concurrent changes that refer to the content of the same file, varying from 1.5 to 22 times more than in CVS reported analysis. These integration rates are not equally distributed over the life time of the project but are higher close to release dates. In order to prevent conflicts, close to release dates developers should use awareness mechanisms about the location of their changes. Awareness, defined by [71] as an "understanding of the activities of others which provides a context for your own activity", has been identified by the CSCW community as one of the most important issues in support of remote collaboration. For instance, the awareness approach proposed in [75] could be adopted to provide developers with warning messages concerning concurrent activity and the possibility to consult a list of conflicts. Based on a selected conflict, a user can set watches for concurrently edited elements. For instance, they can be notified when a collaborator has finished editing the element. Another solution could be annotation of concurrent changes [83]. However, this solution is suitable only for centralised version control systems that rely on a server and consists of computing divergence between the project local version of a user and the current state of the project that is saved on the server. Computing divergence in a distributed version control system where there is no central server remains a challenge. Solutions relying on Conflict-free Replicated Data Types such

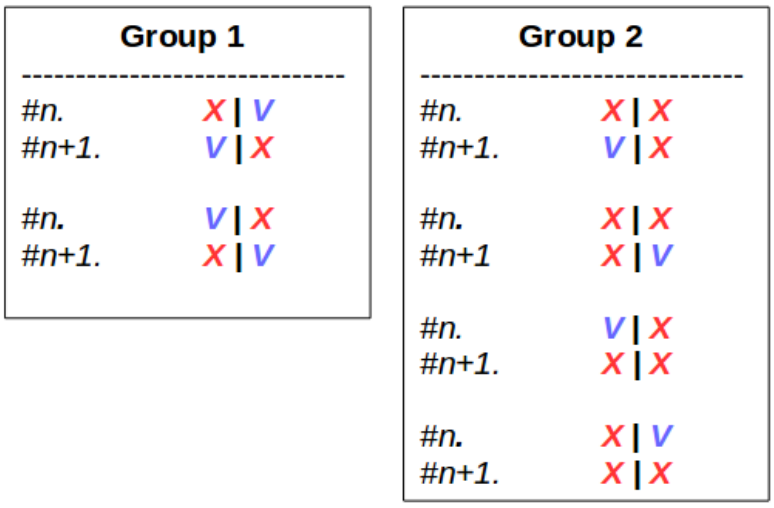


FIGURE 3.13 – Adjacent-line patterns

Project name	*SdMD*	*Lower 95% confidence interval*	*Upper 95% confidence interval*
Rails	1.8872	0.4319	3.6909
IkiWiki	2.0614	1.4932	2.2812
Samba	0.405	-0.0384	0.8483
Kernel	2.1837	1.9854	2.382

TABLE 3.10 – Standardized mean difference(SdMD) effect size between adjacent-line conflicts and normal content conflicts

as [84, 85] that have been adopted by Atom [86] can be used. Moreover, the fact that the change in integration process can be identified based on the integration rate can be of use to other researchers trying to extract process related information from open-source projects.

We found that across the studied four repositories around 75% of the adjacent-line conflicts were false positives. Adjacent-line conflicts detection was designed to warn the developers that there are two changes on adjacent lines that might be related and that developers should check whether the changes are conflicting. For around 25% of cases adjacent-line conflict warnings indeed helped developers to discover the conflicts reducing the costs in later development phase. However, for the other 75% developers did some un-needed extra work for solving the conflicts. Moreover, if concurrent changes occur on two adjacent lines Git signals a conflict, but not in the case of two lines separated by two or more lines. Our results suggest that Git should reconsider signalling conflicts on adjacent lines inside the source code file which requires developers to do in most of the cases some extra work for removing the conflict. A solution would be that Git sends a warning message to the users in the case of concurrent changes on adjacent lines, but does not represent this conflict inside the source code file. Note that in Subversion and Darcs version control systems, concurrent modifications on adjacent lines are not considered as conflict. Our suggestion would be useful for tool builders to help developers in avoiding wasting time on trivial merge conflicts.

Our study features some pitfalls of mining DVCS repositories as they lack a centralized logging

server. The history may be rewritten by the repository owner [50] : the order of commits can be altered, commits can be removed, a sequence of commits can be flattened from multi-branches into a single branch, edits can be squashed in multiple commits via *rebase*. In our analysis we ignored merges flattened due to rebasing. In [61] the authors presented a new method called *continuous mining*. Instead of mining only the primary repository (called *blessed*), this method continuously observes all known repositories of a software project to cover the complete history of the project development. Their empirical study focuses on Linux Kernel [35] which has 479 repositories (from 2012). Among these repositories, 22% did not contribute a single commit to the *blessed*. Nevertheless, *continuous mining* is still a re-active logging mechanism and it is not considered as a permanent solution to the need of centralized logging features of Git.

By analysing only the history from official servers of four projects, we also missed information regarding user communication. The short description of commits does not include this data. Email threads in project's mailing-list such as Linux Kernel Mailing List [87] and GitHub conversations (comments of a pull-request or an issues) are promising for extracting this data.

```
Input:
     A = [1, X, 3, Y, 5]
     O = [1, 2, 3, 4, 5]
     B = [1, 2, W, 4, 5]
Expected merge result:
     Merge(A,B) = [1, X, W, Y, 5]
```

FIGURE 3.14 – 'Adjacent-line changes' example and expected merge result

A	*O*	*B*	*merge status*
1	1	1	stable
X	2	2	
3	3	W	conflict
Y	4	4	
5	5	5	stable

TABLE 3.11 – Diff3 parse for 'adjacent-line' example

A	*O*	*B*	*merge status*
1	1	1	stable
X	2	2	changed in A
3	3	W	changed in B
Y	4	4	changed in A
5	5	5	stable

TABLE 3.12 – Expected parse for 'adjacent-line' example

We showed that in most of the cases detected adjacent line conflicts were false positives, i.e. they were resolved by keeping both concurrent modifications.

In what follows we investigate the reasons for detection of adjacent line conflicts and we suggest a solution to overcome this limitation. Git uses 'snapshot' storage method and it needs

diff3 algorithm to merge two diverging versions. A technical investigation about Git and *diff3* shows that *diff3* algorithm was implemented correctly for Git [88]. The main issue is whether *diff3* works smoothly in adjacent-line cases. Figure 3.14 presents an 'adjacent-line-changes' example with expected result in which changes are merged successfully. *diff3* takes three inputs (A,O,B) in which A and B are the current versions derived from original version O. A, O and B are presented as lists of lines. Changes were made on different lines in A (the second and the fourth lines) and in B(the third line). *diff3* parse in Table 3.11 shows that 'merge(A,B)' results in conflict. The expected merge result can be achieved only if *diff3* could generate the expected parse in Table 3.12. This raises the question whether the adjacent-line conflict is detected due to the fact that the changed regions by the two sites (A and B) are not 'well separated'.

To answer this question, we conducted another example in which changes are 'well separated' by an unmodified line. Figure 3.15 shows *diff3* input of a 'well separated changes' conflict which is supposed to be merged successfully. In this case, changes made on site A are : delete the first line (1) then move the last three lines (5,6,7) at the beginning of the document which is equivalent to deletion of these lines (5,6,7) and their insertion to the first position. Changes made on site B are : deletion of the third line. All these changes are 'well-separated' with the second and the fourth lines which are unmodified. Table 3.13 shows a conflict at the first chunk of *diff3* parse while the expected result is illustrated in Table 3.14. Similar to the formal investigation of *diff3* [89], this example shows that *diff3* does not guarantee a conflict-free synchronization of changes from two sites even though they are 'well separated' by unmodified regions.

```
Input:
     A = [5, 6, 7, 2, 3, 4]
     O = [1, 2, 3, 4, 5 ,6, 7]
     B = [1, 2, 4, 5, 6, 7]
Expected merge result:
     Merge(A,B) = [5, 6, 7, 2, 4]
```

FIGURE 3.15 – 'Well-separated-changes' example and expected merge result

A	*O*	*B*	*merge status*
	1	1	
	2	2	conflict
	3	4	
	4		
5	5	5	
6	6	6	stable
7	7	7	
2			
3			changed in A
4			

TABLE 3.13 – Diff3 parse for 'well-separated-changes' example

Overall, we claim that *diff3* itself cannot be used to generate the expected results for modifications done on adjacent-lines. Moreover, *diff3* is not 'stable' as it relies too much on *two-way-diff*'s outputs which could be biased [89]. Instead of depending only on *diff3*, Git can take advantage

A	*O*	*B*	*merge status*
5 6 7	1	1	changed in A
2	2	2	stable
3	3	0	changed in B
4	4	4	stable
	5 6 7	5 6 7	change in A

TABLE 3.14 – Expected parse for 'well-separated-changes' example

of 'changeset' storage method. Note that Darcs uses 'changeset' and SVN uses both 'changeset' and 'snapshot' storage methods and they can merge successfully concurrent changes done on adjacent lines.

3.4 Chapter conclusion

The goal of this work was to analyse concurrencies and conflicts in Git, a decentralized version control system. We analysed the corpus of four large projects in Git and ours findings are as follows.

Generally, a higher *integration rate* of a project does not generate a higher *unresolved conflict rate*. It depends on how the integration process is managed. Linux Kernel is a large-scale project with a list of subsystem maintainers which keep the lowest *conflict rate* for this project although its *integration rate* is much more higher than the other projects. Excepting Linux Kernel, Rails which was developed using GitHub pull-based model has 5 times lower *conflict rate* than Samba(shared repository) and 3 times lower than IkiWiki(private repository). A more detailed analysis based on release dates shows that each project has its own integration process with dynamic integration rates that changed during project's development cycle.

The rollback action is used more often in case of higher-order conflicts than in case of textual-conflicts. Most of rollback actions do not create new commits (*level=0*) meaning that users generally try to test the merge result before deciding to rollback or not.

In contrast to Darcs and SVN, Git considers changes made on two adjacent lines as content conflict. Based on our analysis, users mostly apply all changes when resolving the adjacent lines conflicts. We proposed that Git should merge concurrent changes on two adjacent lines and throw a warning message instead of considering them as conflicting.

Chapter 4

Time-position characterization of Conflicts in Collaborative Editing

4.1 Introduction

In this chapter we present our study about time-position characterization of conflict in collaborative editing using ShareLaTeX, a real-time web-based collaborative editor.

Most of the early researches of Collaborative Editing are based on 'interviewing' people who have participated in some collaborative editing works. They tried to figured out the essential features of Collaborative editing and how to support them by the mean of computer software (also known as 'Group ware'). According to the 'Taxonomy of Collaborative Editing' presented by Posner et al [3] and extended by Lowry et al [4], Collaborative editing includes five important components which are *roles, activities, document control methods, writing strategies and work-modes*. The 'role' and 'activities' focus on the 'subject', i.e the authors. The 'document control method' focuses on the 'object', i.e the document. The 'writing strategies' focuses on the 'process' of creating document. And the 'work-mode' component, also known as 'Time Space taxonomy') [7], describes the physical distance between collaborators (i.e 'space' dimension) and the synchronicity of editing activities (i.e 'time' dimension).

Early researches of Collaborative Editing also suggest the requirements for designing tools (groupware, collaborative editor) that support collaborative editing [7, 3]. They all agree that Collaborative tools (collaborative editors) should support both 'asynchronous' and 'synchronous' work-modes. Besides, collaborative tools should allow authors to switch easily between two two work-modes (i.e smooth the transition between 'asynchronous' and 'synchronous') [71, 6]. There were some collaborative editors that support both 'asynchronous' and 'synchronous' work-modes such as GROVE (a textual multi-user outlining tool) [7], ShEdit (a multi-user text editor) [21]. However, they were used for research purpose only (i.e used inside laboratory for academic experiments of CE).

Recently, real-time online collaborative editors such as Google Docs [22], ShareLaTeX [23], Etherpad [24] are popular with many useful features to support collaborative editing such as adding comments, in-line communication (chat), revision histories and editing logs. In these tools, changes from one author are almost visible immediately to other co-authors. At each side, changes are 'transformed' on top of other changes so that all authors have the same state of the document. They make it easy for users to edit collaboratively with each other.

CSCW researchers are interested in 'How people edit together now' when collaborative tools make it easy for users to edit simultaneously [29, 90]. Beside the traditional 'interviewing' people

that have participated in some 'collaborative editing sessions' approach, CSCW researchers can also analyse the logs of collaborative editing retrieving from real-time online collaborative editors. According to our literature, Birnholtz et al [68, 30] were the first research that consider to analyse CE logs which were collected and parsed from the Google Docs revision histories. Their studies are strongly controlled by the authors. They include two separated phases : asynchronous writing and synchronous writing. The authors focused uniquely on analyzing the relationship between communication and editing as well the collaborator's social relationships.

There are some other researches that are based on analysing the logs of collaborative editing activities to study how people edit in collaboration. Sun et al [28] presented a detailed analysis of activities logs over two years of all Google employees using Google Docs suite. They found that collaboration editing has grown rapidly up during the period they examined and 'concurrent editing is sticky' (i.e the employees who participate in a 'concurrent session' will do so again in the following month). Olson et al [29] examined the traces of collaborative writing behavior of advanced undergraduates in a project course using Google Docs. The authors focused on assessing the quality of document and analysing different aspects of CE using the taxonomy of CE [3, 4]. D'Angelo et al [31] analysed the histories of a large collection of documents edited in Etherpad[24] to study how people write in the wild. They found that simultaneous editing happens very rarely within a predefined *'time-position window'*.

In general, the process of collaborative editing can be split into several *'session'* of editing which are performed by a single author or several authors. They are denoted as *'single-authored session'* and *'co-authored session'* respectively. This fragmentation process requires a predefined 'interval' [28, 69] or 'maximum time gap' [29] which is not yet well defined in previous studies. Besides, a detailed analysis of editing activities inside sessions, specially *'co-author sessions'* which probably contain *'collaborative edits'*, can bring us valuable insight of collaborative editing sessions. Having a closer look at the cases in which two or more co-authors edit very close to each other : in the same place of the document at the same time (with some predefined 'time-position extension'). These cases can potentially become conflicting cases. It's interesting to see how people manage these potentially conflict cases. In this study, we examined different 'maximum time gaps' to see how choosing the 'maximum time gap' affect to the splitting document into sessions. We then study the different editing behavior between 'single-authored' sessions and 'co-authored' sessions. And finally, we also study two 'potential conflicting cases' in which two co-authors edit very close together in both 'time' and 'position' dimensions.

The rest of the chapter is organized as follows. In section 4.2, we present some related works which are based on analysis of the traces of collaborative editing. In the next section 4.3, we describe the measurements of our study. We also present the algorithms that are used in our analysis in section 4.4 (Algorithm section). And finally, we discuss about our results and conclusion of this study.

4.2 Related work

In this section we present some recent studies on analysis of logs of collaborative editing.

Sun et al [28] published an in-house study that analysed the logs of activity for all Google employees in two years from 2011 to 2013. They found that on that period, the percentage of new employees who collaborate on Docs per month has risen from 70% to 90%. To estimate the percentage of documents which had concurrent editing, they used a *15 minutes interval* to split documents into intervals and consider edits by different users in the same *15 minutes intervals* as concurrent edits. The choice of *15 minutes intervals* is arbitrary. And this approach has edge

cases in which two users edit the same document within 15 minutes but they are split into two adjacent intervals and are not counted as concurrent edits. In addition, they also proposed a more accurate approach which is looking for a sequences of edits by different users with the maximum gaps 15 minutes. However they did not apply it.

Olson et al [29] collected and analysed 96 Google Docs documents written by 32 teams of undergrad students from the Project Management class in three successive years (2011, 2012 and 2013) at University of California, Irvine. They found that 95% of the documents exhibited some simultaneous work. In fact, they used the approach which is similar to the proposed one of [28] with the *7 minutes gaps*. To determine the *7 minutes gap*, they examined all documents with *15 minutes gap* and found that 90% of them were 7 minutes or less. In more details, a document consists of many *sessions*. Each *session* combines a series of *slices*. Each *slice* which aggregates a series of keystrokes is generated after a certain of pause or a certain amount of edits. If a *session* was edited by more than one editor, they consider it as a *simultaneous session*. The others are considered as *solo-authored sessions*.

Both studies above focus only on time-dimension of collaborative editing. If two authors edit a document within 7 or 15 minutes gaps, they are considered as having a simultaneous writing session either they can edit in adjacent positions or far different positions. The choices of 7 or 15 minutes gaps are still arbitrary. In another study, Larsen et al [91] use a mixed methods involving interviews and analysis of the traces of collaborative editing documents (using Google Docs) to outline the role of 'territorial functioning' in CE. On their analysis, they take into account the position-dimension of edits to visualize the 'editing territories' of different authors over the time. However, for the time-dimension, they use the same technique as previous studies of Sun et al[28] and Wang et al [69] which is based on the *15 minutes time gap*.

D'Angelo et al [31] presented a study on how Etherpad, a real-time collaborative editing tool, is used in the wild. They analysed the histories of a large collection of documents (about 14000 pads) in both time and position dimensions. Each edits are independently classified as collaborative or not in time, position and time-position. An edit is considered as collaborative in time dimension if it is *closed enough in time to an edit applied by a different author*. In this study, they used the *time windows* which are 5, 10 and 60 seconds to determine if an edit is *closed enough* or not. And similarly, they used the *position window* which are 10, 80, 400 and 800 characters for position dimension. For two-dimensional analysis (time-position), all pairs of *time/position windows* were used. As the results, they found that about half of the pads were edited by a single author. Asynchronous collaboration in which users edit in closed positions of the document but in different times happens often. Simultaneous editing in which users edit in closed positions within the same *time window* happens very rarely. Note that they used the proportion of time, position and time-position collaborative edits over the total edits of the documents for their inferences.

While [28] and [29] focus on finding if a document has some *simultaneous sessions* or not, [31] focuses on finding the quantity of *simultaneous edits* of shared editing documents. It presents a more detail quantitative analysis of *collaborative editing* than the two previous works. However it lacks the overview of how people work together. For example, people can use 'divide and conquer' strategy in which each editors works in different parts (positions) of the document [69]. Then it's obviously that the document presents only *time collaborative edits* results on their analysis. Beside, the *5 seconds or 10 seconds time window* is too short to have multiple editing activities. People can stop to discuss or to read the work of the others in several minutes before continue to write. Moreover, when people are free to collaborate, they do not edit simultaneously all the time. There are several *sessions* that they work asynchronously [29].

Besides, with researches that use data from collaborative editing in Google Docs [28, 29],

although the document revision histories provided by Google Docs capture the details of edits at keystroke level, they were usually grouped into a 'slice' and assigned a single time-stamps. It means that a series of insertions and/or deletions which are performed in a short period have the same time-stamp. This prevents a more fine-grained analysis of editing actions in a short time period. Besides, they focus only on 'time dimension' to determine whether an edit is performed in 'collaboration' (i.e 'collaboration edit') with others or not.

4.3 Measurements

We analysed [23] logs which were collected from a ShareLaTeX server used inside an Engineering school and anonymized for privacy purpose. The logs were conducted from collaborative editing of groups of three or four students which were assigned a writing task and required to use ShareLaTeX server hosted by the Engineering school. All editing activities were recorded by ShareLaTeX server from the beginning until the end of the assignment (from 20-September-2017 to 20-November-2017). Students were notified that they can work in collaboration using ShareLaTeX.

In ShareLaTeX, there are two types of edit which are 'Insertion' and 'Deletion'. They are recorded with following information : the *time-stamp* when they happen, the *position* in the document where they happen, the *user-id* who performs the edit, the *action-type* which determines it is an 'Insertion' or a 'Deletion', the *content* which is inserted or deleted. The *content* of an edit can be a single character or a long string. In addition, a copy-paste action is considered as an 'Insertion'. A replacing action is considered as a 'Deletion' of the old content following by an 'Insertion' of the new content.

We retrieved 1748 documents from the logs. However, 856 documents were created for testing purpose (i.e they were created and edited by a single user and have none or only one edit action). In the rest 892 documents, only 108 of them has more than one author. As we are focusing on collaborative editing, our analysis was performed in these 108 documents which potentially have some collaborative editing works. Table 4.1 presents the overview of our data in which *'No. of authors'* and *'No. of edits'* are the number of editors and the number of recorded editing activities of each documents. The *'Amount of edit'* is the sum up of the length of all edits's *content*.

	Min	*Max*	*Average*	*Std*
No. of authors	2	4	2.69	0.87
No. of edits	53	38329	8000	10583
Amount of edit	245	272935	47133	56866

Table 4.1 – Overview of the data : 108 documents

A document can be presented in time-position view (two dimensional view). Figure 4.1 presents a sample document which is segmented into three *writing sessions* by time dimension. These sessions are clasified into *single-author-session* (SAS) and *co-author-session* (CAS) depended on the number of editors of each sessions. In this sample we have one SAS and two CASs. Noted that in a CAS, two or more editors can edit in the same position or in different positions. For more details in 'time dimension', we have 'internal time distance' (or *internal-distance*) which is the time distance between two adjacent edits in the same session and 'external time distance' (or *external-distance*) which is the time distance between two adjacent sessions in a document.

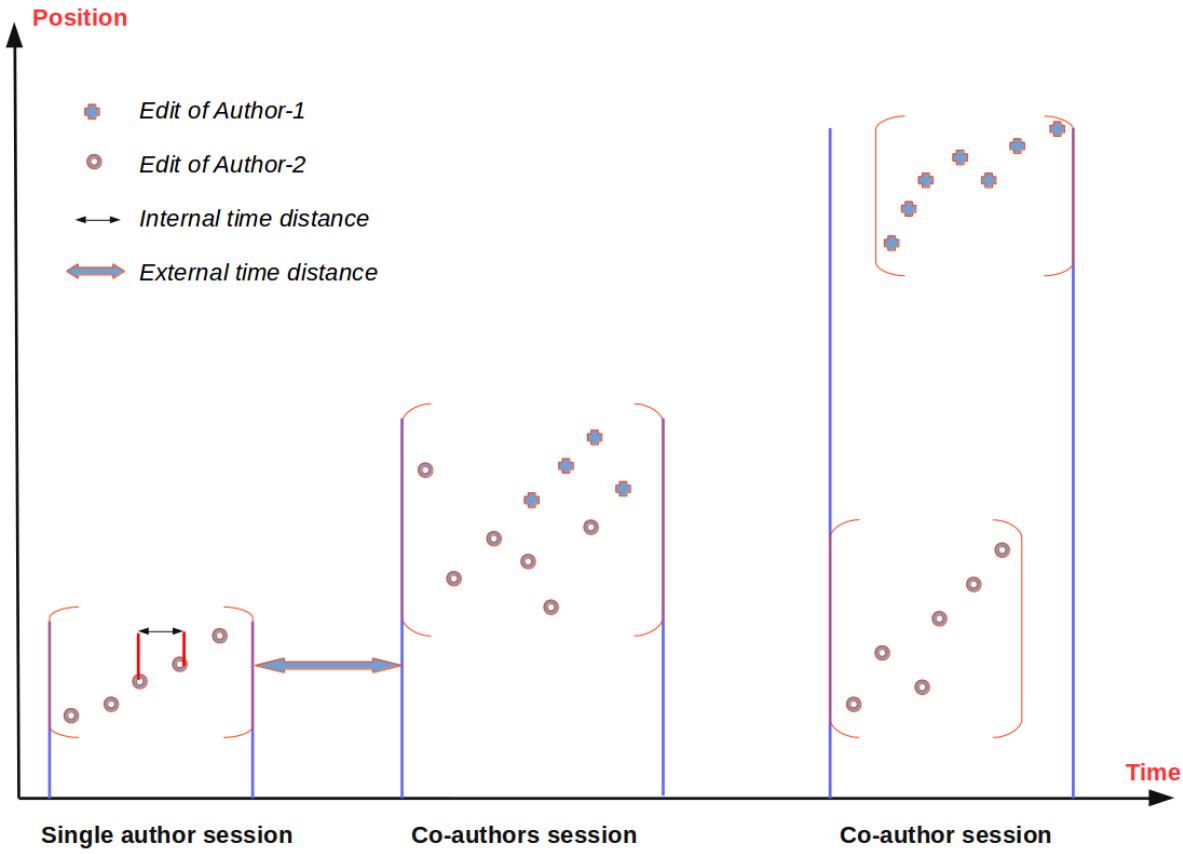


FIGURE 4.1 – A document with two authors in time-position view

4.3.1 Time dimension

We first borrow the proposed approach of [28] to analysis the 'time dimension' of our data. In stead of using only an *'arbitrary maximum time gaps'*, we try to examine the data with different *'maximum time gaps'* : 15 minutes, 7 minutes, 5 minutes, 2 minutes, 1 minute and 30 seconds. Further more, after discrediting a document into sessions and classifying sessions into SASs and CASs, we then analyze the differences between CASs and SASs such as : the internal-distance which is the distance between two edits in the same session, the average time which is the average length of sessions, the average number of edits of sessions.

Table 4.2 presents all the results of our analysis in time dimension. *Doc having CAS(s)* shows the number of documents that have at least one co-author session. The proportion of *Doc having CAS(s)* over all analysed (108) documents is presented in the next line. It's 77.77% with *15 minutes time gap* and downs to 69.44% with *30 seconds time gap*. In comparison to [29] which showed that 95% of documents which exhibited some 'simultaneous work' with *7 minutes time gap*, it's 75.92% in our analysis. Also with this time gap, our analysis shows that the average length of co-author sessions is 2369 seconds (39.5 minutes) and the longest co-author session is 8639 seconds (144 minutes) while they are 9.2 minutes (average) and 74 minutes (longest) in [29].

No. of CASs per Doc is the average number of CASs in each documents after segmenting with the given *time gaps*. The proportion of CASs over all sessions (CASs and SASs) is presents in the adjacent row. The smaller *time gaps* given, the more CASs are generated. However, it reduces the

Time gaps	*15 minutes*	*7 minutes*	*5 minutes*	*2 minutes*	*1 minute*	*30 seconds*
Doc having CAS(s)	84/108	82/108	80/108	77/108	76/108	75/108
Proportion	77.77%	75.92%	74.07%	71.30%	70.00%	69.44%
No. of CASs per Doc						
Average	2.3	2.9	3.4	5.8	9.4	14.1
Proportion	28.4%	24.4%	22.7%	17.5%	13.9%	10.7%
Internal-distance						
SASs (Average)	9.61s	6.77s	5.89s	4.04s	2.87s	2.01s
SASs (CI 99%)	[6.56-12.68]	[5.74-7.80]	[5.19-6.61]	[3.73-4.35]	[2.74-3.00]	[1.97-2.07]
CASs (Average)	4.25s	4.19s	4.15s	2.55s	1.78s	1.24s
CASs (CI 99%)	[2.89-5.62]	[2.69-5.70]	[2.71-5.59]	[1.97-3.14]	[1.54-2.02]	[1.13-1.35]
Session length						
SASs (Average)	972s	647s	507s	226s	109s	51s
CASs (Average)	3314s	2369s	1953s	878s	350s	155s
No. of edits						
SASs (Average)	213	170	146	94	60	39
CASs (Average)	2140	1841	1629	961	470	255
CASs (Normalized)	893	787	704	429	214	118

TABLE 4.2 – Documents segmentation by different *maximum time gaps*

proportion of CASs over the total sessions. In another way, the number of CASs increases slower than the number of SASs when the *time gap* decreases. *Internal-distance* presents the average internal distance between two edits in the same session (SASs or CASs). For more details, we calculated the confidence interval with 99% of significance (CI 99%) for the internal distance variable. We found that the internal distance of SASs is longer than the internal distance of CASs. In another way, the distance between two edits in single-author sessions is longer than the one in co-authors sessions. *Session length* shows the average length in seconds of each sessions, ie. how long each sessions lasts. *No. of edits* is the average number of edits in each sessions. We also found that (with 99% of significance) the average length of CASs is longer than the average length of SASs and also CASs have larger number of edits than SASs in average. Noted that to compare *No. of edits*, we had normalized *No. of edits* of CASs as it includes edits from all collaborators while the one of SASs includes edits of a single editor. Normalized *No. of edits* of each documents is calculated by dividing original *No. of edits* to the number of collaborators. Having more edits with shorter distance between edits and longer collaborating time, it significantly gives us a quantitative view that the co-authors sessions have more contribution to the documents than the single-author sessions.

As the *maximum internal-distance* is limited by the *time gap*, it's obviously that the *internal-distance* presented in 4.2 must be shorter than the given *time gap*. However, the result shows that the average *internal-distance* is much shorter than expected. For a better understanding, we calculated the confidence interval of *internal-distance* for each documents with 90% significance. Both SASs and CASs were included in this analysis. Figure 4.2 presents the results of *420s (7 minutes) time gap* in which all the confidence intervals do not reach 80 seconds and the general average *internal-distance* of all documents is 6.5 second presented by the red line. We can see that the *7 minutes time gap* is not a suitable *time gap* for our corpus. The suitable *time gap* should be shorter than 7 minutes.

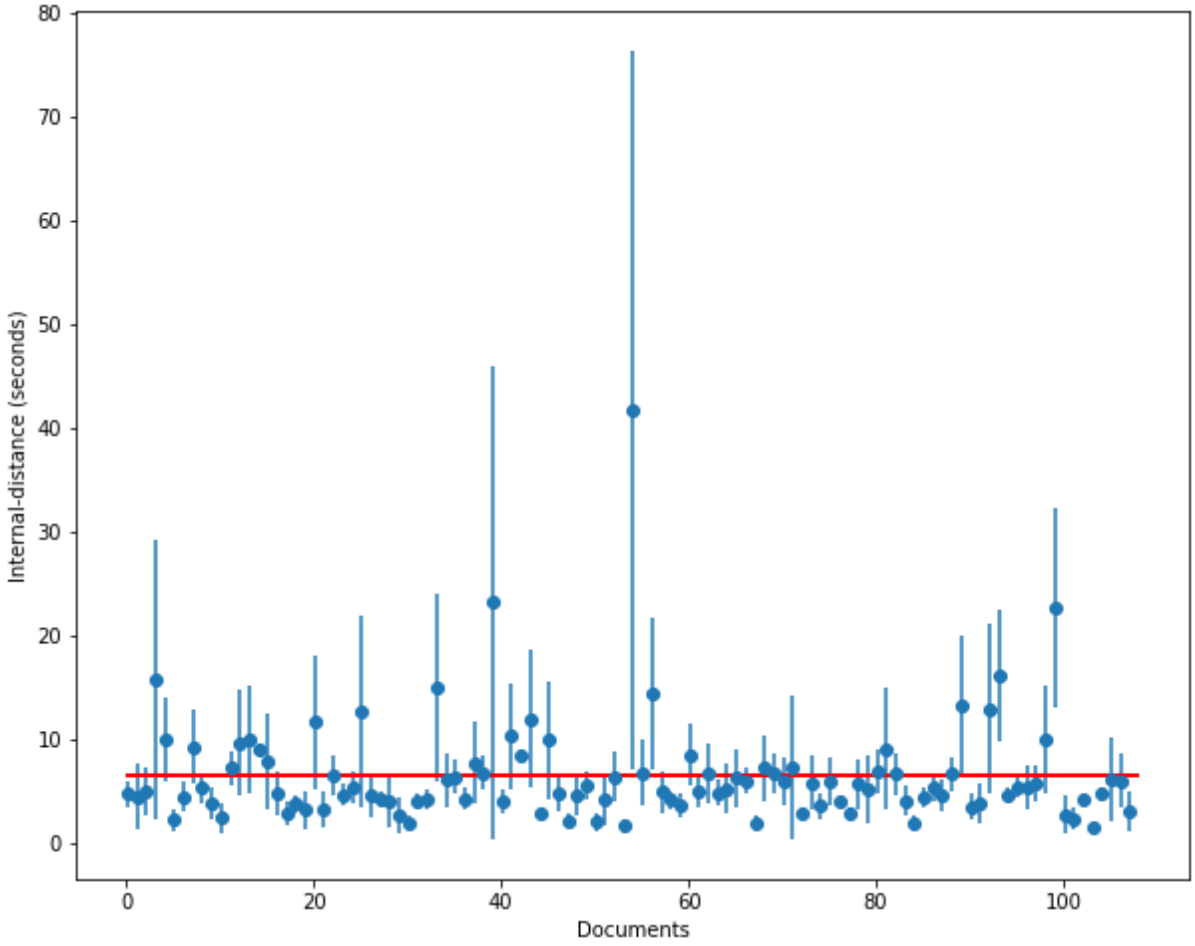


FIGURE 4.2 – Time gap = 420s (7 minutes), Average internal-distance with confidence interval CI90

In addition, we measured the *external-distances* of of *30 seconds time gap* and calculated their distribution in different intervals which are created from *potential time gaps* : [30s,60s) , [60s,120s), [120s,180s), [180s,240s), [240s,300s), [300s,420s), [420s,900s), [900s,). In which the left square bracket '[' denotes *'equal or longer than'*, the right round bracket ')' denotes *'shorter than'* and the last interval denotes the *external-distances* which are *'equal or longer than 900 seconds'*. Our suggestion is that if an interval covers more *external-distances* than others, it has much 'potential' to become a suitable *time gap* than others. Figure 4.3 shows the details distribution of *external-distances* in each documents. In average, they are 39.73 %, 24.58%, 8.98%, 4.37%, 2.53%, 3.03%, 3.98% and 12.80% respectively for the given intervals. In another way, if we increase the *time gap* from 30 seconds to 60 seconds, 39.73% of sessions will become a part of another sessions because *external-distances* can not be shorter than 60 seconds. If we use 120 seconds *time gap* in stead of 30 seconds *time gap*, 64,31 % (39.73% +24.58%) of sessions will be merged into another sessions and so on. From above results, we can say that the [30s, 60s) and [60s,120s) intervals has much potential to contain the suitable *time gap* as it covers much more *external-distances*(64,31%) than others. Moreover, we found that the range of *external-distances* is very wide, from 30 seconds to 87 hours (3.6 days). As our corpus were collected from the writing task in which students had about 2 months to finish their work, they could have multiple writing sessions during this time. Also, students could have several subjects in a semester and this writing task was given by one of them so that they they could work on it only one or two times a week.

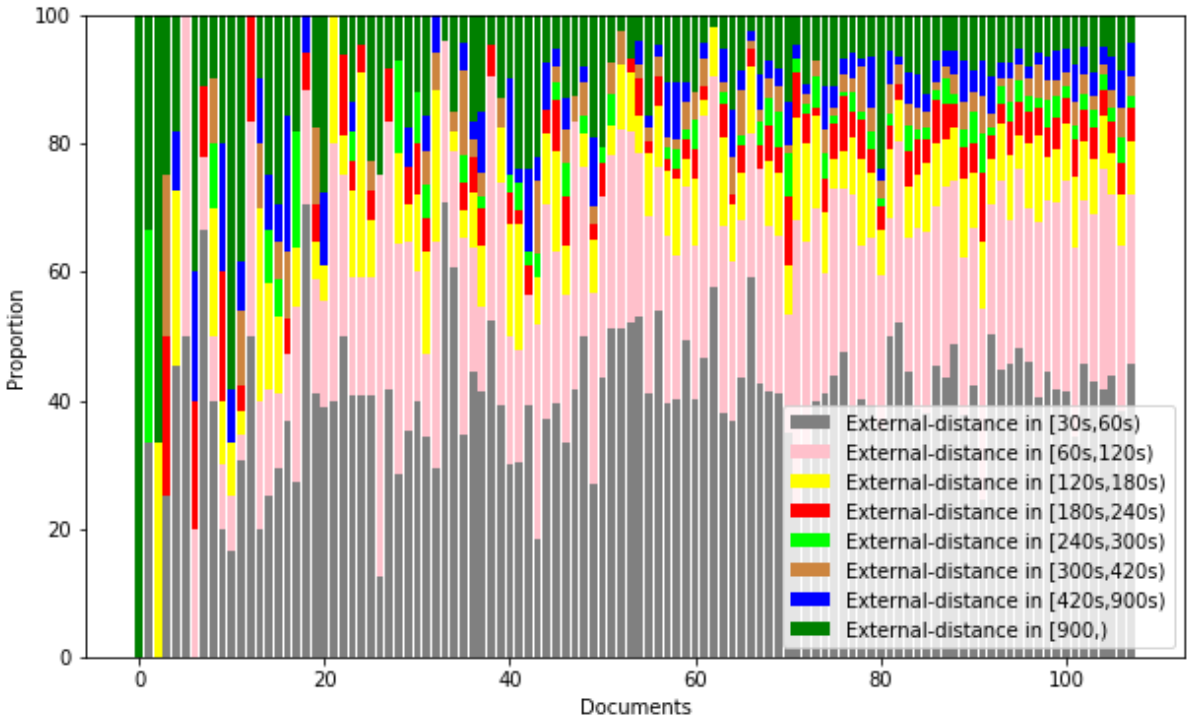


FIGURE 4.3 – Time gap = 30s, External-distances distribution

In summary of time-dimension analysis, we found that collaborative editing is usually separated into many editing sessions including *single-author sessions* and *co-author sessions*. The time distance between sessions has a very wide range (up to 87 hours in our case study). To

split a document into sessions, a suitable *time gap* needs to be determined. And finally, editors have more editing activities in co-authors sessions than in single-author sessions, ie. having more contributions when working collaboratively.

4.3.2 Time-position analysis

The analysis on *time dimension* gives us an overview of how collaborative editing happens over the time. It determines *co-authors sessions* in which the authors write closely together in time. However, it lacks the information about whether or not they write 'closely' in the same part of a document or 'separately' in different parts of it. A more detailed analysis in both time-position dimensions give us a better understanding about how they write collaboratively.

[Time, Position] window	*Time collaborative edits*	*Position collaborative edits*	*Time-Position collaborative edits*
[10s, 80c]	18.09 %	69.25 %	3.05 %
(CI 99%)	[13.23-22.95]	[62.03-76.49]	[1.59-4.50]
[30s, 400c]	25.04 %	84.64 %	10.70 %
(CI 99%)	[18.86-31.23]	[78.63-90.65]	[6.90-14.49]
[60s, 800c]	30.27 %	90.19 %	17.49 %
(CI 99%)	[23.06-37.48]	[85.09-95.27]	[12.08-22.90]

TABLE 4.3 – Proportion of time/position/time-position collaborative edits by different windows

We first use the same approach of D'Angelo et al [31] which is to classify edits operations as 'collaborative' (i.e close to edits of a different individual in 'time', 'position' or both 'time-position') or 'non-collaborative' to have an overview of the collaborative editing process of our corpus. We also calculate the proportion of time-collaborative edits, position-collaborative edits and time-position collaborative edits over the total number of edits of each document. We focus on three sizes of the 'time-position' window : the small size - *[10 seconds, 80 characters]*, the medium size - *[30 seconds, 400 characters]* and the large size - *[60 seconds, 800 characters]*. In their study, D'Angelo et al also consider a very small 'time-position' window which is *[5 seconds, 10 characters]*. Noted that the time different between the small size and the very small size is only 5 seconds. Table 4.3 presents the proportion of 'collaborative' edits in 'time', 'position' and both 'time-position' over the total number of edits of each document. For the small size window ([10s,80c]), the proportion of 'time', 'position' and 'time-position' collaborative edits are 18.09%, 69.25% and 3.05 % respectively. They are 11,28% , 49.65%, 6,62% respectively from the results of the corpus of D'Angelo et al. For the large size window ([60s,800c]), the proportion of 'time', 'position' and 'time-position' collaborative edits are 30.27%, 90.19% and 17.49 % respectively. They are 21.92% , 61.17%, 18.71% respectively from the results of the corpus of D'Angelo et al. The medium size window ([30s, 400c]) is not considered in the work of D'Angelo et al so we can not compare. However, with the small and larger size windows results, we have the same findings as them. That's the 'position collaboration' is the most popular and the 'time-position collaboration' happens very rare. The 'position collaboration' refers to two edits of two different users which are close or belong to the same part of a document. The 'time-position collaboration' refers to two edits of two different users which are close to each other in both 'time' (i.e happening within a time-window) and "position' (i.e their position-distance is smaller than a position-window). Figure 4.4 illustrates this finding in a more clearly way. We can see

that the proportion of 'position collaborative edits' is significantly higher than 'time collaborative edits' and 'time-position collaborative edits'.

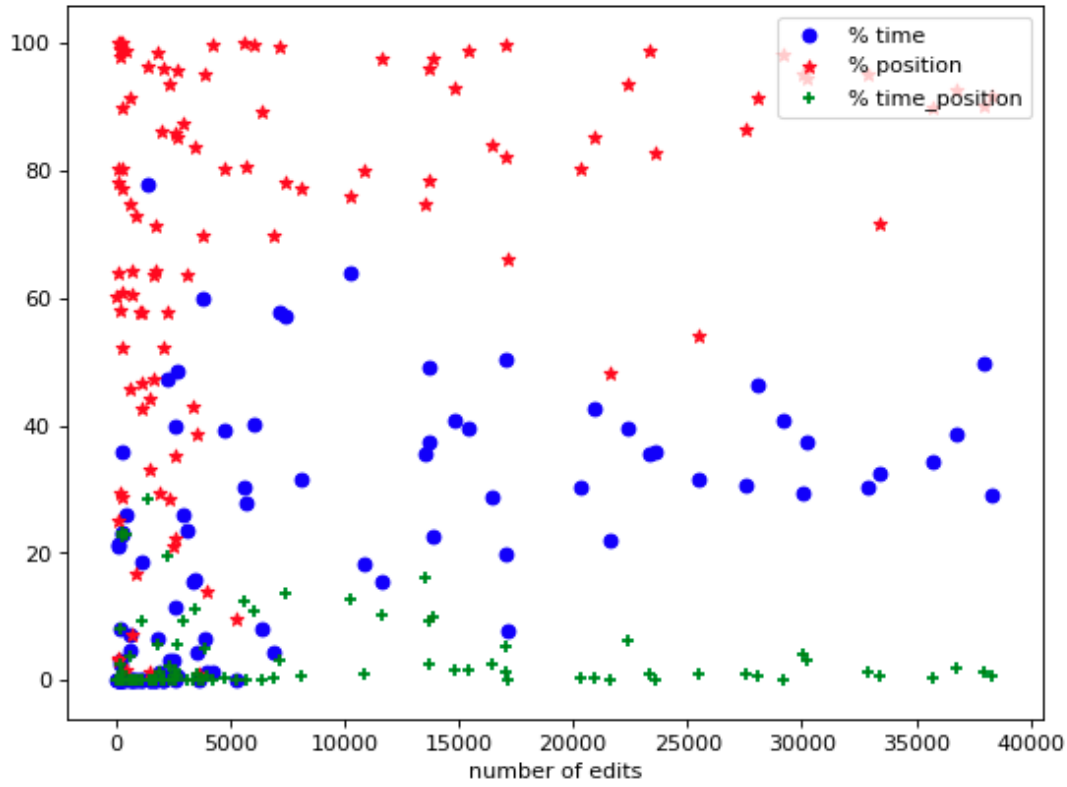


FIGURE 4.4 – Proportion of time, position and time-position collaborative edits over the total number of edits of each document - [10 seconds, 80 characters] window

Above approach considers all edits in the process of collaborative editing as a single session. However, the process of collaborative editing is usually split out into several *'session'* of editing [29]. Although this approach takes into account both time and position dimensions, it still can not give us a precise overview of how people edit together.

As we can see in Figure 4.1, edits inside a session can be grouped into different 'clusters' depending on their time-position distances. We borrow a real document in our data corpus which id is *59e8f8d98e96ef7e2dc01eb2* to explain our time-position analysis. Figure 4.5 presents this document in time-position view with session and and cluster annotations. A session begins with a vertical blue line and ends with a vertical red line. A cluster is presented as a rectangle filled with light-blue color. Edits are presented as orange and green dots depend on which authors they belong to (called orange author and green author). This document contains about 2500 characters (including while-spaces, empty lines) and was edited in total 6000 seconds by 2246 edit actions (including deletion, insertion). In the first 60 minutes, it was edited by the orange author only. Those edits are split into four single-author sessions which the longest time distance between them is about 16 minutes. In the next 25 minutes, it's a co-authors session in which the document was edited collaboratively by two authors (the orange author and the green author). And in the last time, it's a single-author session of the orange author. In this session, the orange author had edited in three different positions of the document with position-distance larger than 400 characters. Noted that in Figure 4.5, for the simple presentation, we use a large size windows

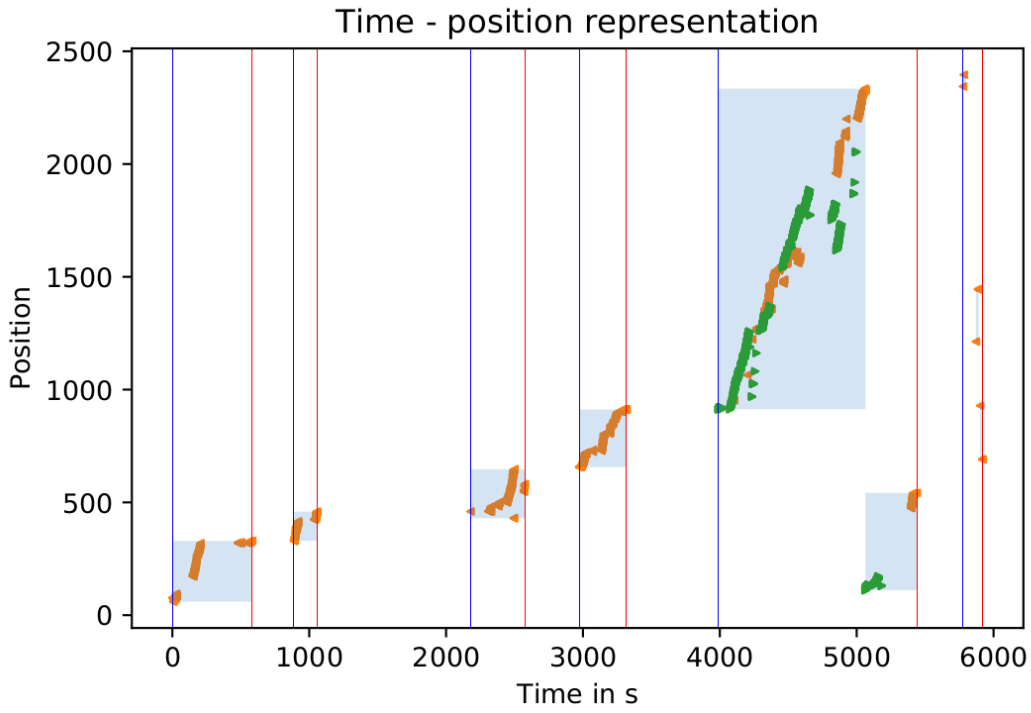


FIGURE 4.5 – A real document(id=59e8f8d98e96ef7e2dc01eb2) presented in time-position view

which is *[time-gap, position-gap] = [300seconds, 400 characters]* to reduce the number of sessions and clusters.

Edits in a single-author session can be re-edited by another author in another session. However, these two sessions are separated in time so that if conflicts happen, they are asynchronous conflicts. In this analysis, we focus on the cases that two or more authors edit closely together in both time and position. Having a closer look in the co-author session in Figure 4.5 which contains two clusters of edits, we can see that these two clusters have edits of both authors. In the bottom right cluster, there is a clear border between edits of two authors. In the top left cluster, beside three borders separating edits of different authors, there are several cases in which one author edited between two continuous edits of another author. This lead to the question that is : can conflict happen in these cases if all involved edits are very close to each other in both time-position dimensions ?

To answer the above question, we examined two cases in which conflicts potentially happen. The first case called *border case* refers to the switch-point between two adjacent editing-areas which belong to two different authors. Those editing-areas can contains one or more continuous edits. The second case called *insertion case*, refer to the case in which one author try to edit between two continuous edits of another author. In another way, one author tries to insert one or more edits into an editing-area of another author. In both cases, if the time-position distance between those continuous edits is small, conflicts have high potential to happen.

Figure 4.6 present the illustration of *border case* and *insertion case*. On the left side of Figure 4.6, **X, Y** and **X'** are three continuous edits (in time order) of two different authors in which **[X, Y]** form a *border case*. This *border case* can become a *potential border conflict case* if its time-position distance defined by the red rectangle is small. And if the adjacent edit **X'** happens between the positions of **X** and **Y**, it should be recommended as a *border conflict*. A formal

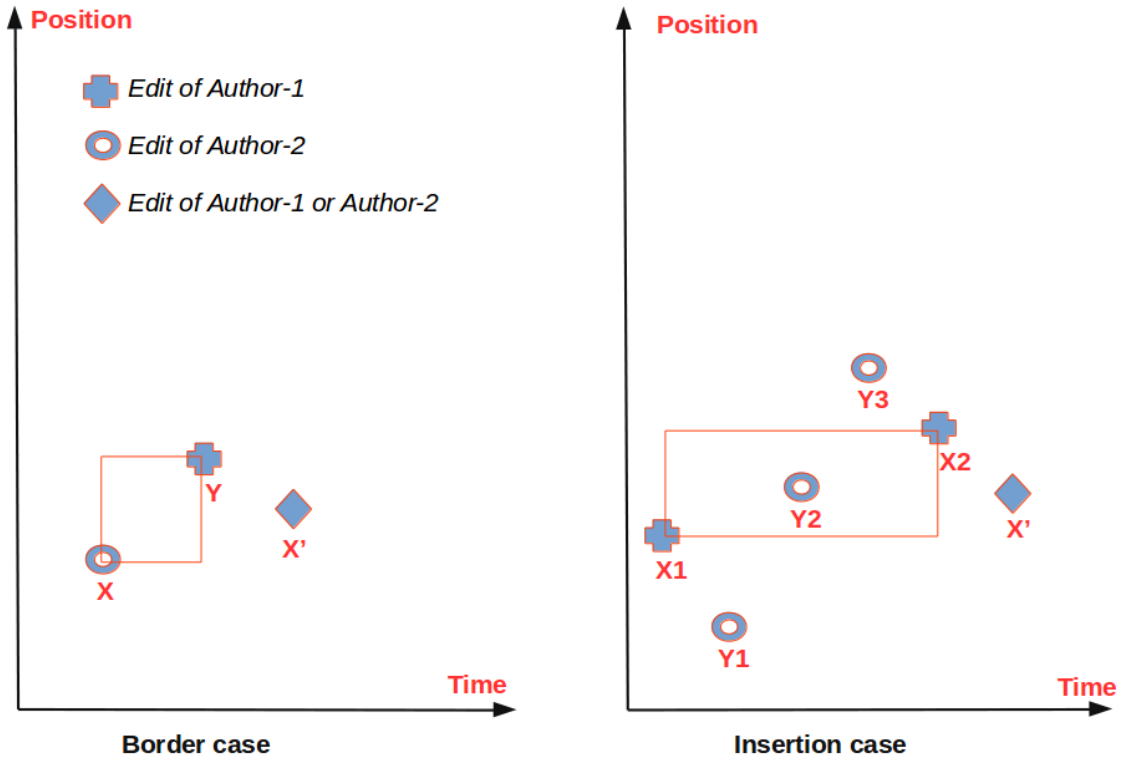


FIGURE 4.6 – Illustration of Border case and Insertion case

description of *border conflict* is presented in Definition 3.

On the right side of Figure 4.6, $[X_1, Y_1, Y_2, Y_3, X_2, X']$ is a sequence of edits in time order. It means that Y_1, Y_2, Y_3 happen between X_1 and X_2 and X' happens right after X_2. However, in position dimension, the order is different, which is [Y_1, X_1, Y_2, X', X_2, $Y3$]. Focusing on $[X_1, Y_2, X_2]$, we can see that they satisfy both time order and position order. In another way, author2 had inserted an edit into the time-position window formed by two continuous edits of author1. In the Figure 4.6, this time-position window is presented by the red rectangle created by X_1 and X_2. If this window is small and the adjacent edit **X'** happens between the position of X_1 and X_2,we consider this case as an *insertion conflict*. A formal description of *insertion conflict* is presented in Definition 4.

Definition 1. *A sequence of edits* $[X_1, X_2, .., X_n]$ *is a* ***sequence of edits in time order with time-gap t*** *if* $\forall i \in [1, n) : time(X_{i+1}) > time(X_i)$ *and* $time - distance(X_i, X_{i+1}) < t$

Definition 2. *A sequence of edits* $[X_1, X_2, .., X_n]$ *is a* ***sequence of edits in position order with position-gap p*** *if* $\forall i \in [1, n) : position(Xi + 1) > position(Xi)$ *and* $position - distance(Xi, Xi + 1) < p$

Definition 3. *X is an edit of Author-1, Y is an edit of Author-2 and X' is an edit belong to one of them. If* $[X, Y, X']$ *is a sequence of edits in time order with time-gap t and* $[X, X', Y]$ *is a sequence of edits in position order with position p,* $[X, Y]$ *then form a* ***border conflict*** *within time-position window [t,p].*

Definition 4. X_1, X_2 *are edits of Author-1,* $Y_1, Y_2, ...Y_k$ *are edits of Author-2 and X' is an edit belong to one of them.* $(Y_i)^+, i \in [1, k]$ *is a sub-sequence of edits of Author-2 which has at least*

one edit. If $[X_1, Y_1, Y_2, ...Y_k, X_2, X']$ *is a sequence of edits in time order with time-gap t and* $\exists(Y_i)^+$ *so that* $[X_1, (Y_i)^+, X_2]$ *or* $[X_2, (Y_i)^+, X_1]$ *form a sequence of edits in position order with position p,* $[X_1, (Y_i)^+, X_2]$ *or* $[X_2, (Y_i)^+, X_1)$ *then form an* ***insertion conflict*** *within time-position window [t,p].*

We use a **[30s, 10c]** time-position window to run our experiments. As we had explained in Section 4.3.1, the documents will be separated into sessions using a *30 seconds time-gap*. After that, all co-authors-sessions are checked for *border cases* and *insertion cases*. If these *border cases* and *insertion cases* satisfy the selected time-position window which is **[30s, 10c]**, they become *potential border conflicts* and *potential insertion conflicts*. And if one of involved authors edits right after in the potential-conflict-area, we consider that potential conflict as a conflict. The reason that we choose the **[30s, 10c]** time-position window is that it has enough for authors to have three or more editing actions and can cover the position distance of two or three words. The detailed algorithms are presented in 'Algorithm' session (session 4.4). Table 4.4 presents the result of **[30s, 10c]** time-position window.

	Border conflict	**Insertion conflict**
Proportion of **Potential-conflicts** over Consider-cases	5.7%	2.27%
CI99%	[1.73-9.66%]	[0-5.04%]
Proportion of **Conflicts** over Potential-conflict	84.52%	97.22%
CI99%	[77.53-91.51%]	[88.96-100%]
Average of **Time-distance** of Conflict cases	6.17s	4.06s
CI99%	[3.67-8.68s]	[0-10.3s]
Average of **Position-distance** of Conflict cases	3.43c	4.15c
CI99%	[2.88-3.98c]	[2.13-6.17c]

TABLE 4.4 – Border conflict and Insertion conflict with **[30s, 10c]** time-position window

The results in Table 4.4 shows the high proportion of potential-conflicts that become conflicts. It's from 77.53% to 91.51% for *border conflict* and from 88.96% to 100% for *insertion conflict* with significance of CI99%. However, these two types of conflict happen very rarely. It's less than 9.66% for *border conflict* and less than 5.04% for *insertion conflict*. In case of *potential border conflict* that is not a *border conflict*, it means that the time-position window (the border) of two continuous edits of two authors is larger than the time-position window that we use to determine *border conflict*. And in the case of *potential insertion conflict* that is not a *insertion conflict*, it means that two authors are editing in two different areas which are large enough in position.

Beside the time-position window *[30s, 10c]* that we use in above experiment, we also use a smaller and a bigger window to examine the *border conflict* and the *insertion conflict*. The results of using a smaller and a larger time-position windows which are *[10s, 5c]* and *[60s, 20c]* are presented in Table 4.5 and Table 4.6.

	Border conflict	**Insertion conflict**
Proportion of **Potential-conflicts** over Consider-cases	3.07%	2.13%
CI99%	[0.9-5.23%]	[0-5.49%]
Proportion of **Conflicts** over Potential-conflict	84.04%	100%
CI99%	[76.93-91.16%]	[NA]
Average of **Time-distance** of Conflict cases	2.59s	4.34s
CI99%	[1.51-3.68s]	[0-12.91s]
Average of **Position-distance** of Conflict cases	2.23c	2.3c
CI99%	[1.88-2.57c]	[0.98-3,62c]

TABLE 4.5 – Border conflict and Insertion conflict with **[10s, 5c]** time-position window

	Border conflict	**Insertion conflict**
Proportion of **Potential-conflicts** over Consider-cases	9.94%	2.04%
CI99%	[3.95-15.93%]	[0-4.33%]
Proportion of **Conflicts** over Potential-conflict	87.0%	95.53%
CI99%	[80.98-93.02%]	[85.05-100%]
Average of **Time-distance** of Conflict cases	5.33s	4.24s
CI99%	[3.16-7.5s]	[0-9.14s]
Average of **Position-distance** of Conflict cases	7.46c	4.9c
CI99%	[5.96-8.97c]	[2.41-7.39c]

TABLE 4.6 – Border conflict and Insertion conflict with **[60s, 20c]** time-position window

From the results we can see that the *potential border conflict* is affected by the time-position window more than the *potential insertion conflict*. The *potential border conflict* decreases from 5.7% to 3.07% with a smaller window and increases to 9.94% with a larger window. The *potential insertion conflict* has less effects by the size of time-position window. Further more, the smaller time-position window decreases the proportion of *border conflict* over *potential border conflict* while the larger window increases it. For the *insertion conflict*, it's reversed. In another way, with

smaller window, the *potential insertion conflict* is more likely to become *insertion conflict*.

4.4 Algorithms

In this section, we describe the algorithms which are used in our analysis. Except for the first two functions calculating 'time-distance' and 'position-distance' between two given edits *e1 and e2* presented in Algorithm 1, other algorithms require the list of edits of a document sorted by 'time' in ascending order. We denote this list as *'E'*. As we mentioned in Chapter 4, edits of all documents are retrieved from the logs of ShareLaTeX, grouped by documents which they belongs to and sorted in ascending order by their time stamp (i.e 'time').

4.4.1 The calculation of 'time-distance' and 'position-distance'

Recall that all operations from clients are 'transformed' (i.e serialized) before being applied and stored at the server [92]. It means that for all edits which are retrieved from the logs stored at the ShareLaTeX server, their 'position' are actually transformed. If the timestamp of edit *e1* < the timestamp of edit *e2*, *e1* is applied before *e2*. And of course the 'position' (i.e the offset) of *e2* is already transformed based on *e1*. When retrieving all edits of a document from the logs of ShareLaTeX server, we sort them by their 'timestamp' (i.e by the order that they were applied to the document). We then calculate the 'time-distance' and 'position-distance' as follow :

Algorithm 1: Calculating time-distance and position-distance

```
   Input: two edits e1, e2
   Output: time-distance t
1  Function Time-distance(e1, e2) :
       // e2.time > e1.time
2      return e2.time - e1.time
3
   Input: two edits e1, e2
   Output: position-distance t
4  Function Position-distance(e1, e2) :
       // e2.time > e1.time
5      if e2.position > e1.position then
6          return e2.position - e1.position
7      else
8          if e2.action-type == 'INSERTION' then
9              return e1.position - e2.position + e2.length
10         else
11             return e1.position - e2.position - e2.length
12
```

The *'time-distance'* between two edits *e1* and *e2* in which *e1* 'happens' (is applied) before *e2* is simply calculated by the time difference of *e1* and *e2*.

The *'position-distance'* between two edits *e1* and *e2* in which *e1* 'happens' (is applied) before *e2* depends on the positions between *e1* and *e2*. We take into account two cases in which the first case (Case1) is *position (e2) is larger than position (e1)* and the second case (Case2)

is *position (e2) is less than position (e1).* The **'+/- length(e2)'** operator indicates that the 'length(e2)' is added or subtracted from the 'position-distance' depending on the action-type of *e2.* If *e2* is an 'insertion', 'length(e2)' is added and if *e2* is a 'deletion', 'length(e2)' is subtracted. For example, given sequence S = 'abc' and edit $e1 = insert(1, `XY')$, after applying $e1$, we have $S$ = '$aXYbc'$. If edit $e2 = insert(0, `MN')$, we have S = '$MNaXYbc'$ and $position - distance(e1, e2) = 1 - 0 + 2 = 3$. If edit $e2 = insert(4, `MN')$, we have S = '$aXYbMNc'$ and $position - distance(e1, e2) = 4 - 1 = 3$.

According to the explanation from ShareLaTeX's 'document-updater' repository [92], the length of the previous operation (i.e *e1*) does not affect the offset of later operations (i.e *e2*). It means that we do not need to 'shift' (i.e re-calculate) the 'position' of later operations. And if the position of *e2* is greater than the position of *e1*, the later operation also does not 'affect' the previous operation as the insertion or deletion is performed downward from the position of *e2.* In this case, the 'position-distance' is simply the subtraction of position(e2) to position(e1). However, when the position of *e2* is less than the position of *e1*, we need to consider the length of *e2.* If *e2* is an 'insertion' with 'length = n', the real 'position-distance' between *e2* and *e1* is extended by 'n'. In another way, *e1* is 'shifted' down by 'n' characters (distance unit). It's similar for the case in which *e2* is a 'deletion'. Note that in this case, the 'position-distance' can be a negative number if the 'length' of deletion action (i.e *e2*) is larger than the subtraction of the positions of *e1* and *e2.* It indicates that *e2* operation deletes over the content at position(e1).

4.4.2 Time, Position and Time-Position collaborative edits classifying

To classify edits into 'time'(and/or 'position',and/or 'time-position') collaborative or not, we use the same approach as D'Angelo et al [31]. An edit is considered as a 'time collaborative' edit within a given 'time-extension' (time-window) if there is at least one edit of another user which is close to it in time. More specifically, suppose that *e1 and e2* are edits of two different users. If *time-distance(e1,e2) <= time-window* then *e1 and e2* are 'time collaborative' edits (to each other). It's similar to 'position collaborative' edits in which we consider the 'position-distance' of *e1 and e2.* For the 'time-position collaborative' edits, we need to consider both time and position dimensions. Note that in this approach, if edit *e1* is 'collaborative' to edit *e2* then *e2* is also 'collaborative' to edit *e1.*

As all edits of the document are sorted ascending by 'time', we just need to 'look forward' to find the 'time collaborative' edit of an edit. For instance, to find edit *e2* that is 'time collaborative' with edit *e1*, we will search in range of [*e1*.position +1, E.size()]. And when we mark **e1** as 'time collaborative' edit, we also mark *e2* as 'time collaborative' edit (i.e 'looking backward').

Figure 4.7 gives an example in which $e1$ and $e2$ are 'time-collaborative' edits. If the algorithm just looks 'forward', only $e1$ is marked as 'time-collaborative' edit because $e2$ is not 'time-collaborative' to $e3$. The algorithm needs to look 'backward' to see whether $e2$ is 'time-collaborative' to its prior edit (i.e $e1$) or not. Algorithm 2 presents this approach to classify 'time collaborative edits. Note that in this approach, for each edit, we save a list of edits which are 'time collaborative' to it. These lists are used later to classify edits as 'time-position collaborative' edits or not.

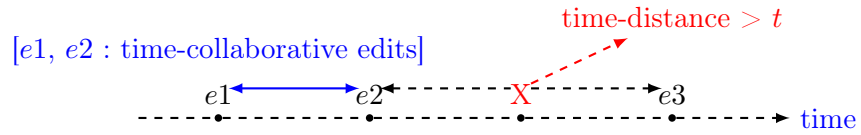


FIGURE 4.7 – An example of 'time-collaborative' edits

Algorithm 2: Time collaborative edits classifying

Input: predefined time window t, list of edits of a documents sorted by 'time' E
Output: list of Time collaborative edits T

```
/* For each edit, we save a list of edits that are 'time collaborative' to it. These lists
   are used to to check if an edit is 'time-position collaborative' or not.            */
1  for each edit e in E do
2      e.time-collaborative = new List()
3
4  for each edit x at index i in (0, length(E) -1) do
       /* Looking forward                                                              */
5      for each edit x2 at index j in (i, length(E) -1) do
6          if Time-distance(x, x2) <= t and x.user <> x2.user then
7              x.time-collaborative.append(x2.index)
               /* Looking backward, i.e x is 'time-collaborative to x2 then x2 is also
                  'time-collaborative' to x                                            */
8              x2.time-collaborative.append(x.index) /* Edit x is already set as
                  'time-collaborative' edit. However, the algorithm need to find all x2s which
                  are 'time-collaborative' with x for 'looking backward'               */
9          else
               /* Time-distance (x,k)> t, ∀k ∈ [x2, xN]                               */
10             Break
11
12 for each e in E, not e.time-collaborative.isEmpty() do
13     T.append(e)
14 return T
```

To find the 'position collaborative' edit of an edit, we need to search through all edits of the document (i.e list of edits *'E'*). The simple Algorithm 3 presents a simple approach with the complexity of $0(n^2)$. It works for small size documents. However, it becomes very slow for the large documents which have more than 40000 edits (run-time is usually up to several hours). An optimized solution is to segment edits of the document into several segments using the 'predefined position window' p. We then need to search in a small subset of list of edit *'E'*. Algorithm 5 illustrates this optimized approach using 'segment-edits-by-position' function which is presented in Algorithm 4.

Algorithm 3: Position collaborative edits classifying - Complexity = $O(n^2)$

Input: predefined time window p, list of edits in a document sorted by 'time' E
Output: list of Position collaborative edits P

```
1  for each edit e in E do
2      e.position-collaborative =-1
3
4  for each edit x in E do
5      for each edit x2 in E do
           /* Position-distance(x1, x2) assumes that x1.time < x2.time */
6          if x.time < x2.time then
7              if Position-distance(x, x2) <= p and x.user <> x2.user then
8                  x.position-collaborative = x2.index
9                  x2.position-collaborative = x.index
10         else
11             if Position-distance(x2, x) <= p and x.user <> x2.user then
12                 x.position-collaborative = x2.index
13                 x2.position-collaborative = x.index
14
15 for each e in E, e.position-collaborative != -1 do
16     P.append(e)
17 return P
```

The 'Segment-edits-by-position' function (Algorithm 4) gets a predefined position distance and the list of edits sorted ascending by time as its inputs (denoted as p and E). It traverses through the list of edits and build segments of edits based on the 'position-distance' of two adjacent edits. For example, $e1$ belongs to segment $s1$ and $e2$ is the following edit of $e1$. If the position distance of $e1$ and $e2$ is smaller than or equal p then $e2$ also belongs to segment $s1$. If the position distance of $e1$ and $e2$ is larger than p then the function will create a new segment $s2$ and $e2$ belongs to $s2$. This approach has an edge case in which users edit in several areas which position distance are larger than p in one session - called 'Z-case'. In 'Z-case', two or more edits which are close in positions can belong to different segments. Figure 4.8 illustrates a 'Z-case' in which the algorithm creates four segments instead of two segments (marked with blue rectangles). To solve this edge case, we use 'Merge-close-segment' (Algorithm 4) function which builds a new segment from 'close' segments.

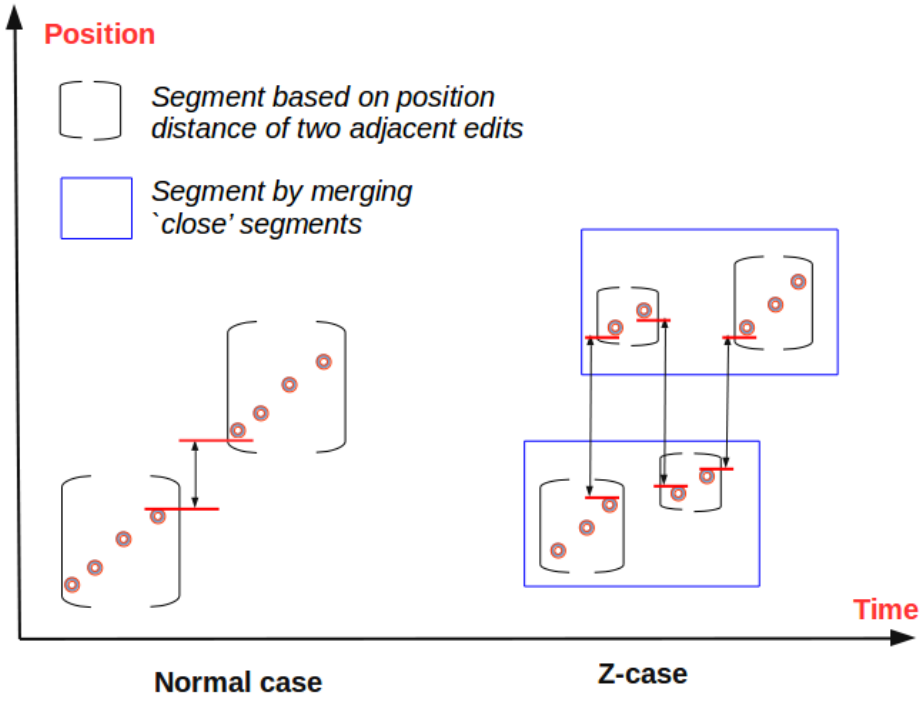


FIGURE 4.8 – Segment edits by position - Z-case

Algorithm 4: Segment-edits-by-position

```
   Input: edit e, segment s, predefined position window p
   Output: TRUE if e belongs to segment s , FALSE if not
 1 Function Belong-to-segment(e, S, p) :
 2     minPos = edit in segment S which has smallest position
 3     maxPos = edit in segment S which has largest position
 4     if e.position > maxPos.position then
 5         return Position-distance(e, maxPos) < p
 6     else if e.position < minPos.position then
 7         return Position-distance(e, minPos) < p
 8     else
 9         return TRUE
10
   Input: list of segments of edits S
   Output: new list of segments of edits S
11 Function Merge-close-segment(S) :
12     S2 = new List()
13     for each segment s with index i in list of segment S and s.merge = FALSE do
14         newSegment = s
15         s.merge = TRUE
16         for each segment s2 with index j= i+1 in list of segment S and s2.merge =
            FALSE do
17             if is-close(newSegment, s2) then
18                 newSegment.append(s2)
19                 s2.merge = TRUE
20         S2.append(newSegment)
21     return S2
22
   Input: predefined position window p, list of edits E
   Output: list of Segments of edits S (segmented by position using window p)
23 Function Segment-edits-by-position(p, E) :
24     tempSegment= new List()
25     for each edit e in E do
26         if tempSegment.size() = 0 then
27             tempSegment.append(e)
28         else if Belong-to-segment(e,tempSegment, p) then
29             tempSegment.append(e)
30         else
31             S.append(tempSegment)
32             tempSegment = new List()
33     S= Merge-close-segment(S)
34     return S
```

Algorithm 5: Position collaborative edits classifying - Using Segment-edits-by-position()

```
   Input: predefined position window p, List of edits E
   Output: P : List of Position collaborative edits
1  Function Classify-position-collaborative-edits(p, E) :
2      for each edit e in E do
3          e.position-collaborative =-1
4
5      S = Segment-edits-by-position(p,E) ;
6      for each segment s in list of segments S do
7          for each edit x in segment s do
8              for each edit x2 in s, x <> x2 do
                   /* Position-distance(x1, x2) assumes that x1.time < x2.time   */
9                  if x.time < x2.time then
10                     if Position-distance(x, x2) <= p and x.user <> x2.user then
11                         x.position-collaborative = x2.index
12                         x2.position-collaborative = x.index
13                 else
14                     if Position-distance(x2, x) <= p and x.user <> x2.user then
15                         x.position-collaborative = x2.index
16                         x2.position-collaborative = x.index
17
18     for each segment s in list of segments S do
19         for each edit e in segment s, e.position-collaborative != -1 do
20             P.append(e)
21     return P
```

An edit which is 'time-position collaborative' to another edit is also 'time collaborative' to that edit (and of course, also 'position collaborative' to it). So to classify a pair of edits as 'time-position collaborative' or not, we check only the edits that are already marked as 'time collaborative'. Remember that in the 'Time collaborative edits classifying' algorithm (Algorithm 2), we did save a list of 'time collaborative' edits for every edits. If an edits is 'time collaborative' edit, its list is not 'null'. These lists are used in 'Time-Position collaborative edits classifying' algorithm (Algorithm 6).

Algorithm 6: Time-Position collaborative edits classifying

Input: predefined time-position window t, p, list of edits E
Output: C : List of Time-Position collaborative edits

```
/* Get all edits that are 'time collaborative' within time window t                    */
T = [For each e in E and e.time-collaborative != 'NULL']
for each edit x in T do
    for each edit x2 in (x.time-collaborative) do
        /* x and x2 is already close in time (i.e 'time collaborative' to each other, so
           x.user <> x2.user)                                                           */
        if Position-distance(x, x2) <= p then
            x.time-position-collaborative = x2.index
            x2.time-position-collaborative = x.index
for each e in E, e.time-position-collaborative != -1 do
    C.append(e)
return C
```

In 'Time', 'Position' and 'Time-Position' collaborative edits classifying algorithms, we consider the whole document as a big session of collaborative editing. We implement these algorithms to have an overview about our data corpus, in comparison to the work of D'Angelo et al [31]. In addition to previous works, the authors take into account the 'position' dimension with different position window sizes in their analysis. Note that in this approach, if edit *e1* is 'collaborative' (in 'time'/'position'/ 'time-position') to *e2* , we also have *e2* is 'collaborative' to *e1*. Assume a 'time collaborative' scenario in which *user1* makes n sequential edits in a short time. Meanwhile, *user2* also makes m sequential edits in a short time. With some *predefined time window 't'* we can have maximum $n * m$ 'time collaborative' edits in this single 'time collaborative' scenario which is probably counted as one 'co-authors session' in other approaches such as Sun el al [28] or Olson et al [29]. Besides, the proportion of collaborative edits ('time', 'position', 'time-position') is calculated by the number of collaborative edits over the number of all edits of the analyzed document. A document which has many short 'co-authors sessions' and also some long 'single author session' (i.e having a large number of 'non-collaborative' edits) may end up with low 'time collaborative' proportion. And if users use 'divide and conquer' strategy (i.e users have their own 'territory' of writing), it's obviously that the proportion of 'position' (and also 'time-position') collaborative edits very low.

Algorithm 7: 'Potential border conflict' detection

```
   Input: co-authors sessions ca
   Output: list of border-cases borderCases
1  Function Get-list-of-border-cases(ca) :
2      borderCases = new List()
3      for each adjacent edits e1 and e2 in co-author session ca do
4          if e1.user <> e2.user then
5              bCase= new bCase()
6              bCase.time-distance = Time-distance(e1, e2)
7              bCase.position-distance = Position-distance(e1, e2)
               /* to check for correction                                  */
8              bCase.Xid = e1.id
9              bCase.Yid = e2.id
10             borderCases.append(bCase)
11     return borderCases
12
   Input: border case b, co-authors sessions ca, time t, position p
   Output: if border-case b has been corrected after a short time t, return TRUE , else
           return FALSE
13 Function Check-correction-of-border-case(b, ca, t, p) :
14     e1 = ca.get-edit-by-id(b.Xid)
15     e2 = ca.get-edit-by-id(b.Yid)
       /* within position extension p                                      */
16     if b.position-distance <= p then
           /* within time extension t (noted that e.time > e2.time > e1.time) */
17         for each edit e in ca, e.id > e2.id , Time-distance(e, e2) <= t do
18             if e.position is between (e1.position and e2.position) then
19                 return TRUE
20     return FALSE
21
   Input: predefined time-position window t, p, list of 'co-authors sessions' A
   Output: list of 'potential border conflict' B
   /* looking for border cases which have been corrected within time-position extension t,p */
22 Function Potential-border-cases-detection(A, t, p) :
23     for each 'co-authors session' ca in A do
24         borderCase = Get-list-of-border-cases(ca)
25         for each border case b in list borderCases do
26             if Check-correction-of-border-case(b, ca, t, p) then
27                 B.append(b)
28     return B
```

4.4.3 'Potential border conflict' and 'Potential insertion conflict' detection

The two algorithms presented in this subsection present our approach in which we consider the analyzed documents as a sequence of 'single-author sessions' and 'co-authors sessions' (Algorithm 7 and Algorithm 9). And instead of calculating the proportion of 'collaborative' edits in each session, we focus on analyzing the cases in which two different users edit 'close' (i.e 'collaborative') to each other within a predefined time-position window. The two algorithms take the list of 'co-authors sessions' of the analyzed document and the predefined time-position window as their inputs.

For 'Potential border conflict' detection (Algorithm 7), we first search all 'border-cases'(i.e the 'switch-point' of two editing areas of two different users). This list then is filtered using a predefined 'time-position' window to verify that only 'border-cases' in which two different users edit 'close' to each other.

For 'Potential insertion conflict' detection (Algorithm 9), we first find all 'insertion-cases' using $position - distance < p$ condition to make sure only 'close' edits are involved. We then filter these cases using the predefined 'time' condition (i.e time dimension of the predefined 'time-position' window). It's a bit more complicate to find the 'insertion-case' than the 'border-case'. Algorithm 8 presents our solution to find all 'insertion-cases' of a co-author session. A more clearly intuitive illustration is presented in Figure 4.9.

In Figure 4.9, we try to find out if author-Y has inserted some edits between two continuous edits $X1, X2$ of author-X or not. Denote Xi as an edit of author-X, Yi as an edit of author-Y, for each pair of adjacent edits $X1Y$ (i.e $X1.time < Y.time$), we have the initial pattern : $pattern =$ '$X1Y'$ or $pattern =$ '$YX1'$. The prior pattern presents for the case in which $X1.position < Y.position$ and the second pattern presents for the case in which $X1.position > Y.position$. Based on the following edits of Y in the current co-author session, we build up the pattern by adding Y or $X2$ to it. If the following edit is an edit of author-Y, we add Y to the pattern. The position of the Y depends on its position in comparison to the position of $X1$. If the following edit is an edit of author-X (i.e $X2$), we add $X2$ to the pattern with respect to the position of $X1$. And then we start to check whether the final pattern contains $X1(Y) + X2$ or $X2(Y) + X1$ or not. If the final pattern contains one of them, it is an 'insertion-case'. Note that $(Y)*$ presents for an empty string or a string of multiple Y and $(Y)+$ presents for a string which has one or several Y.

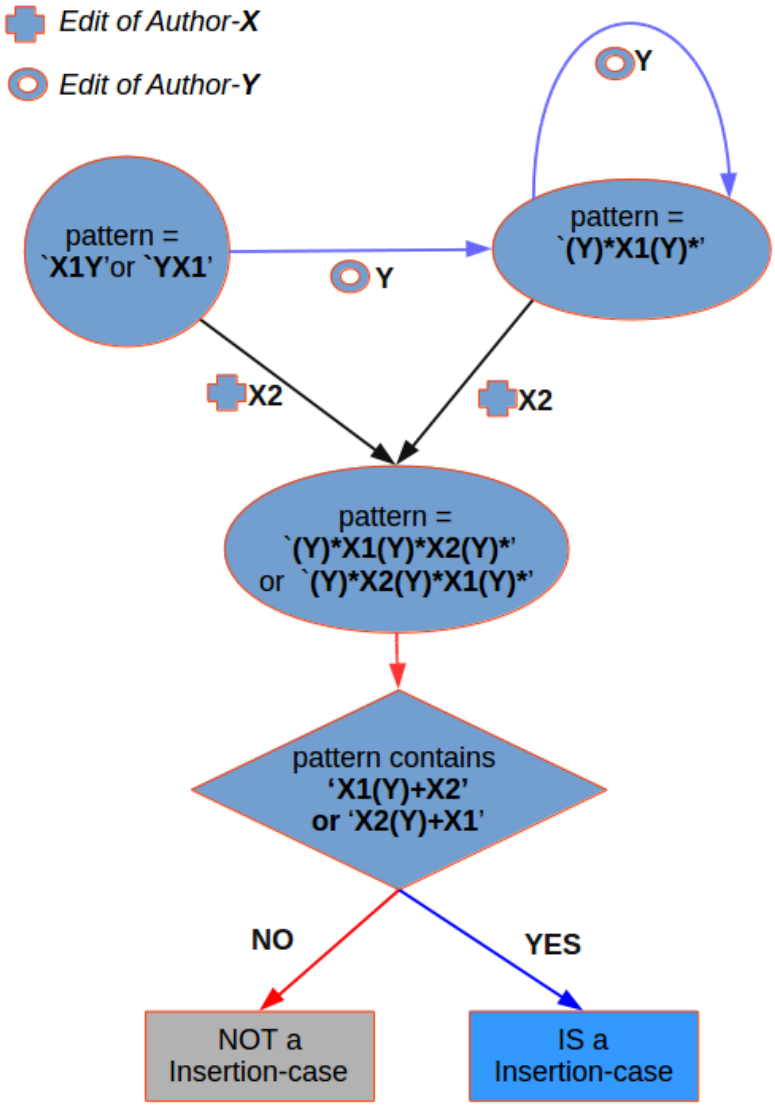


FIGURE 4.9 – Illustration of insertion-case detection (Algorithm 8)

Algorithm 8: Get list of insertion cases

```
Input: 'co-authors sessions' ca
Output: list of border-cases borderCase
1  Function Get-list-of-insertion-cases(ca) :
2      insertionCases = new List()
3      for each adjacent edits X and Y in co-author session ca do
4          if X.user <> Y.user and Position-distance(X,Y< p) then
5              iCase= new iCase()
6              xypattern = ''
7              iCase.begin-id = X.id
8              positionX1= X.position
               /* In case X.position > Y.position, we need to 'shift' the positionX1   */
9              if X.position > Y.position then
10                 xypattern = 'YX1'
11                 positionX1= Shift(Y)
12             else
13                 xypattern = 'X1Y'
14
15             for edit Y2 in ca, Y2.id> Y.id and Y2.user <> X.user and
                Position-distance(X,Y2< p) do
                   /* consider to add 'Y' before or after 'X1'                        */
16                 if Y2.position > Y.position then
17                     xypattern = 'Y'+ xypattern
18                     positionX1= Shift(Y2)
19                 else
20                     xypattern = xypattern + 'Y'
21
22             if Y2.position > positionX1 then
23                 iCase.begin-pos= positionX1
24                 iCase.end-pos= Y2.position
25             else
26                 iCase.begin-pos= Y2.position
27                 iCase.end-pos= positionX1
28
29             iCase.end-id = Y2.id
               /* Find the right position for 'X2' in xypattern                       */
30             indexX2 = Find-position-X2(xypattern.search('X1'), ca[X.id,Y2.id])
31             xypattern= xypattern[ :indexX2] + 'X2' + xypattern[indexX2 :]
32
33             if xypattern.search('X1(Y)+X2') !=-1 or xypattern.search('X2(Y)+X1')
                !=-1 then
34                 insertionCases.append(iCase)
35     return insertionCases
```

Algorithm 9: 'Potential insertion conflict' detection

Input: insertion case in, co-authors sessions ca, time t
Output: if insertion case in has been corrected after a short time t, return $TRUE$, else return $FALSE$

```
1  Function Check-correction-of-insertion-case(in, ca, t, p) :
2      begin = ca.get-edit-by-id(in.begin-id)
3      end = ca.get-edit-by-id(in.end-id)
       /* within time extension t (noted that e.time > end.time > begin.time)  */
4      for each edit e in ca, e.id >end.id, Time-distance(e, end) <= t do
5          if e.position is between (in.begin-pos and in.end-pos) then
6              return TRUE
7      return FALSE
8
```

Input: predefined time-position window t, p, list of 'co-authors sessions' A
Output: list of 'potential insertion conflicts' I

```
   /* looking for insertion cases which have been corrected within time extension t  */
9  Function Potential-insertion-cases-detection(A, t, p) :
10     for each 'co-authors session' ca in A do
11         insertionCases = Get-list-of-insertion-cases(ca)
12         for each border case in in list insertionCases do
13             if Check-correction-of-insertion-case(in, t, ca) then
14                 I.append(in)
15     return I
```

4.5 Chapter conclusion

By examining different 'maximum time gaps' from 30 seconds to 15 minutes we found that the time distance between sessions (i.e *'external-distance'*) has a very wide range (up to 87 hours in our case study) and by evaluating the distribution of *'external-distance'* of a very small time gap, we can determinate a suitable 'maximum time gap' to split editing activities into sessions. In a more detailed analysis of the *co-authors-sessions*, we use a [30 seconds, 10 characters] time-position window to examine the cases in which two authors edit closely together in both time and position. We focus on two cases which potentially result in conflict : *'border cases'* and *'insertion case'*. *'Border cases'* refers the cases in which two different authors edit on the border of two close editing areas that belong to them. And *'insertion cases'* refers to the cases in which one author does some edits between two continuous edits of another author. The results show that these two cases happen rarely : up to 5.04% for *'insertion cases'* and up to 9.66% for *'border cases'*. It means that people rarely edit closely in both time and position. However, these cases (ie. the case in which people edit closely) are more likely to become conflicts. It's from 77.53% to 91.51% of *'border cases'* result in 'conflict' and it's from 88.96% to 100% for *'insertion cases'*. From above results, we suggest that collaborative editing tools (ShareLaTeX in this case) should consider to have an awareness mechanism for these two types of 'potential conflict'.

Chapter 5

Conclusion and Perspectives

5.1 Conclusion

The process of Collaborative Editing is the continuous divergence and synchronization of 'multiple, parallel streams of activity' [11]. Depending on how often the 'synchronization' is performed, the process of CE is classified as 'asynchronous CE' (CE using 'asynchronous' work mode) or 'synchronous' CE (CE using 'synchronous' work mode). A CE process includes several editing sessions (or editing phases) and users are free to use either 'asynchronous' or 'synchronous' work mode in each session.

The thesis includes two studies of conflicts in CE which were structured into two parts :

— Analyzing conflicts in 'asynchronous' work mode in software projects based on Git version control system (Chapter 3).

— Analyzing conflicts in 'synchronous' work mode in documents created in ShareLaTeX (Chapter 4).

In the first part, we used the commit history of four large Git-based open-source projects (Rails [32], Iki Wiki [33], Samba [34] and Linux Kernel [35]) to conduct our analysis. In Git, users can edit in parallel in their own repository and synchronize (integrate) manually. Changes made by a user are visible to other collaborators only when he/she integrates his/her works to the 'official' repository. The integration process may result in conflict. We conducted an analysis of the integration (presented by *'Integration rate'*) and conflict (presented by *'Conflict rate'*) of each project over the total development time and also in specific time periods which are four weeks before and four week after the release date. We found that *'Content-conflict'* is the most popular type of conflict in Git-based projects. And in general, *'Integration rate'* and *'Conflict rate'* have no or very week 'correlation'. In addition to previous works ([15, 16]) we presented a more fine-grained analysis of conflicts. A notable finding is about how people resolve *adjacent-line conflicts*. Around 75% of adjacent-line conflicts were 'false positives' (i.e users resolve them by applying changes from both sides). Our suggestion is that Git should consider to send a warning message to users in an 'adjacent-line' case instead of marking it as a conflict. In addition, we found that users also resolve conflicts by 'roll-back' to previous version. Users usually use 'roll-back' right after the merge. We also reveal in some cases, users use 'roll-back' even though Git merges their changes successfully. These cases can be explained by the assumption that users get some complex 'higher-order' conflicts (i.e compiling or testing failed).

In the second part, we analyze the collaborative editing traces which are collected from a ShareLaTeX server in an Engineering School. Differently from Git, ShareLaTeX integrates edits from all users automatically in real-time. Edits of a user are visible to other collaborators almost

immediately. However, users do not work in 'synchronous' work mode all the time. There are periods when only one author edits the document (called 'single authored' session) and there are periods when two or more authors edit together (called 'co-authored' session). Our corpus includes 108 documents which have more than one author. Each document includes many writing 'sessions'. The distance between two adjacent sessions (denoted as *'external-distance'*) has a very wide range, from 30 seconds to 87 hours. By evaluating the distribution of *'external-distance'*, we can determinate a suitable 'time gap' to split editing activities into different editing sessions. By investigating editing activities inside each session, we found that in general, co-authored sessions have more editing activities than single-authored sessions, the distance between two adjacent edits in co-authored sessions is shorter than in single-authored session and co-authored sessions is longer than single-authored session. In another way, people are more 'productive' in co-authored sessions than in single-authored sessions. In a detailed analysis of 'co-authored sessions', we examined two cases in which conflicts potentially happen : 'border case' and 'insertion case'. 'Border cases' refer to the cases in which two different authors edit in the border of two adjacent editing areas (that belong to them). And 'insertion cases' refer to the cases in which one author does some edits between two continuous edits of another author. The results show that these two cases happen rarely : up to 5.04 % for 'insertion cases' and up to 9.66 % for 'border cases'. It means that people rarely edit closely in both time and position. However, these cases are very likely to become conflicts : 77.53% to 91.51% of 'border cases' and 88.96% to 100% for 'insertion cases' result in 'conflict'. From the above results, we suggest that collaborative editing tools (ShareLaTeX in this case) should consider to have an awareness mechanism for these two types of 'potential conflicts'.

5.2 Publications

- Nguyen, H.L., Ignat, C. Parallelism and conflicting changes in Git version control systems. IWCES'17 - The Fifteenth International Workshop on Collaborative Editing Systems , Feb 2017, Portland, Oregon, United States [93].

- Nguyen, H.L., Ignat, C. An Analysis of Merge Conflicts and Resolutions in Git-Based Open Source Projects. Computer Supported Cooperative Work (CSCW), volume 27, 741–765 (2018). [94]

- Nguyen, H.L., Ignat, C. Time-Position Characterization of Conflicts : A Case Study of Collaborative Editing. In : Nolte A., Alvarez C., Hishiyama R., Chounta IA., Rodríguez-Triana M., Inoue T. (eds) Collaboration Technologies and Social Computing. CollabTech 2020. Lecture Notes in Computer Science, volume 12324, 65–80 (2020). Springer, Cham [95].

5.3 Perspectives

5.3.1 Analysis of CE's traces of Git-based projects

Investigation of parallel development based on version control systems usually comes with the main question : 'How frequently do conflicts occur during integration process and how do users resolve them ?'. In different to centralized version control system such as CVS in which a server records all changes of users, Git - decentralized version control system - does not support the centralized logging feature. The commit history in the official repository of a project is the most 'complete' corpus that we have. To extract useful information such as which merges resulted in conflicting or which files were edited in parallel and needed integration, we need to re-integrate

all developer's changes of a repository and record all outputs of this integration process.

In our study, we have analyzed different types of conflicts which happen in different periods of the project's development (i.e conflicts in different development phases). We also focused on how people integrate their changes during 'active time' which was close to the release dates. However, a more particular insight of each project is still needed. It would be interested and handy to a project manager to know which 'areas' (i.e packages, modules, files) have high rate of conflicts or which developers often commit conflicts or which developers have high conflicting rate. To obtain these information, the analysis should focus also on conflicting frequency of sub-groups (it's usually a 'pair' of developers) who have more common editing objects (i.e file or set of files) between them than with others.

Regarding the 'validity' of our data corpus, the commit history of the official repository, it's obvious that our data does not contain all commits from all 'clone' repositories of the project. It contains only commits that are successfully selected [50]. Commits that are not selected, for example, commits from a pull-request that was rejected, may exist only on some individual's repository but do not appear in the commit history of the official repository. A solution for this problem is to 'fetch' all commits from all 'clone' repositories into a 'super-repository'. Unfortunately, it's not easy to find all 'clone' repositories of a project. Biazzini et al [96] use the list of 'fork' repositories of the official repository provided by GitHub while Gousios et al [64] find new 'clone' repositories by tracking where commits come from. Both of them can not find all 'clone' repositories of a project. In another approach, Pietri et al [97] defined two different 'fork' types in addition to the basic 'forge fork' type (i.e explicit fork - type 1) which are used to identify 'fork' repositories hosted in different platform (hosting service). They are 'shared commit fork' (type 2) and 'shared root directory fork' (type 3). In their empirical analysis using GHTorrent [64] and Software Heritage [66] datasets, they found a substantial amount of 'fork' type 2 (+9% forks) and type 3 (+37%) which are not recognizable as type 1.

Besides, not all 'clone' repositories contribute the official repository. For instance, 25 % of 'clone' repositories of Linux-Kernel project never contributed to its official repository [64]. This approach could expose some bias in the project history retrieved from an 'official repository'. However, it's costly and may potentially create redundancy data. For example, a developer creates a new branch in his local repository (i.e a 'clone') to test his idea. He works on that branch for a while time and plans to delete it later. In this case, if the 'super-repository' fetches changes from this 'clone', it actually fetches valueless commits. In our opinion, the commit history of the official repository is the 'valid' corpus as it contains all main changes that contributors want to share with others. The 'missing' part is the changes that their owners do not want to share yet or plan to delete them later.

5.3.2 Higher-order conflicts and roll-back action

Beside the 'adjacent-lines' conflict of which we suggest Git to send out a warning message, we are also interested in 'semantic' conflicts. Unfortunately, it's difficult to detect semantic or higher-order conflicts in large projects such as Samba or Linux Kernel. Building and testing a larger project are time consuming. Besides, not all projects provide an 'automatic build-scripts' to build the project after a merge and also a 'test suit' to test it. Recently, 'Continuous integration/continuous delivery' (CI/CD) which usually comes 'Continuous testing' gives us an easier way to verify whether a rollback is caused by build-failed or test-failed. In addition to checking all roll-back actions, we propose that we should analyse other communication channels of the project such as mailing-lists, commits messages... in order to understand the story behind each roll-back action. It's interested but also time-costly.

Currently, we assume that the roll-back action applied to a successful merge (i.e without any textual conflict) is caused by a 'higher-order conflict'. It can be a compiling error or a failed test case. It can be also a decision of the author or the leader, i.e the author wants to dismiss the merge because of some reason. Mining mailing list of open-source projects could be a promising approach to reveal the real reason of each roll-back action. Note that each project has a different developers team, so the analysis can be very specific to a particular project.

5.3.3 Time-position analysis of CE's logs collected from real-time web-based collaborative editors

Sun et al [28] used a fix 'time interval' to split edits of a document into several *15 minutes sessions* while Olson et al [29] used the 'maximum time-gap' approach to split a collaborative editing process into different editing sessions. These sessions are classified as *single-authored* session or *co-authored* session depending on whether they have been edited by more than one user or not. As a result, we know if a document has some *'co-authored sessions'* or some *'simultaneous works'* or not. This approach is quite simple and its result gives us a basic overview about the documents. It's good for preliminary analysis, preparing, pre-processing data for further analysis. Also, it's important to choose a suitable 'maximum time-gap'.

D'Angelo et al [31] used a different approach in which edits are classified as 'collaborative' in 'time', 'position' and 'time-position' or not. Classifying 'position collaborative' edits is time-costly , especially for a document which has a large number of edits (the complexity is n^2 where n is the number of edits). It also gets a heavy bias when users use 'divide and conquer' writing strategy (i.e each user writing only on his/her assigned part). From our perspective, this approach is more suitable to use in the detailed analysis of *'co-authored'* sessions than the whole collaborative editing process (i.e the whole edits of a document).

We suggest that we should use the 'maximum time-gap' approach first, to find out all *co-authored sessions*. We then need a more detailed analysis of these *co-authored sessions* such as classifying 'time-position collaborative edits' for each *co-authored sessions* or detecting 'potential border and insertion conflicts'.

5.3.4 Potential conflicts in real-time collaborative editing

As a design implication of our study, we recommend that an awareness mechanism, which provides to authors information about detected potential conflicts (of both types : border conflicts and insertion conflicts), should be implemented. And then, when an author writes closely in time and position with other authors, they can get notified if potential border conflicts or potential insertion conflicts is likely to happen in the next edit. For example in the border-case, if its time and position distances satisfy the predefined 'time-position window' (i.e its potential border-conflicting case), the awareness system should send out a warning message to both authors so that they should change their editing behavior to avoid the next conflicting edit.

In fact, most of current real-time collaborative editors support multiple cursors with a distinguished color for each user. If two users edit close together, they are aware of the appearance of another cursor in their editing area. People usually avoid making changes in this 'potential conflicting' cases [91]. If one of them continues to edit in a 'potential conflicting' case, the other one usually stop editing there and navigate to another part. However, users must comeback later to check and re-edit (if needed) this 'potential conflicting' area. A stored list of all 'potential conflicting' positions (areas) based on 'border conflict' and 'insertion conflict' will be very handy in these cases to remind users after a period of time. It's also important to determine by mean

of user studies when should the system remind users to re-check (re-visit) a 'potential conflict' area that they previously edited.

Besides, our proposed algorithms for detection of potential conflict (both 'border conflict' and 'insertion conflict') depend on a list of edits which are sorted by timestamp. This list is actually the sequence of edits made by different users over the time. Assume that we schedule the collaborative editor to run these detecting algorithms repeatedly with an interval *interval*. It's obvious that the following executed time can inherit from the previous executed time. Let $L1$ be the list of edits of a document at time $t1$ and $L2$ be the list of edits of that document at time $t2 = t1 + interval$. New coming 'potential conflicts' mostly come from edits made during the *interval* between $t1$ and $t2$. It could have some side effect if we choose a fixed *interval* instead of a flexible *interval*. For example, if we choose a fixed *'15 minutes interval'* (i.e the algorithms are scheduled to run every 15 minutes), two potential edits could belong to two different executions, so the algorithm can not detect them as 'potential conflict' case. To avoid this side effect, we should use a flexible *interval* based on the *'maximum time gap'*. That is we only run the detection algorithms when users stop editing for a period of time that is longer than a predefined *'maximum time gap'*. Two edits that belong to two different executions are not 'time collaborative' (time different is large than *'maximum time gap'*) so that they are not 'potential conflict' edits.

Appendix A

Proportion of time, position and time-position collaborative edits

This annex includes three figures that present the proportion of 'time collaborative' edits, 'position collaborative' edits and 'time-position collaborative' edits over the total number of edits in a document. For each document, these proportion are presented by the blue circle, the red start and the green cross respectively. Three different 'time-position windows' were used are : [10 seconds, 80 characters] (Figure A.1), [30 seconds, 400 characters] (Figure A.2), [60 seconds, 800 characters] (Figure A.3). They all show that 'position' collaboration happens more often than 'time' and 'time-position' collaboration. And 'time-position' collaboration happens very rarely.

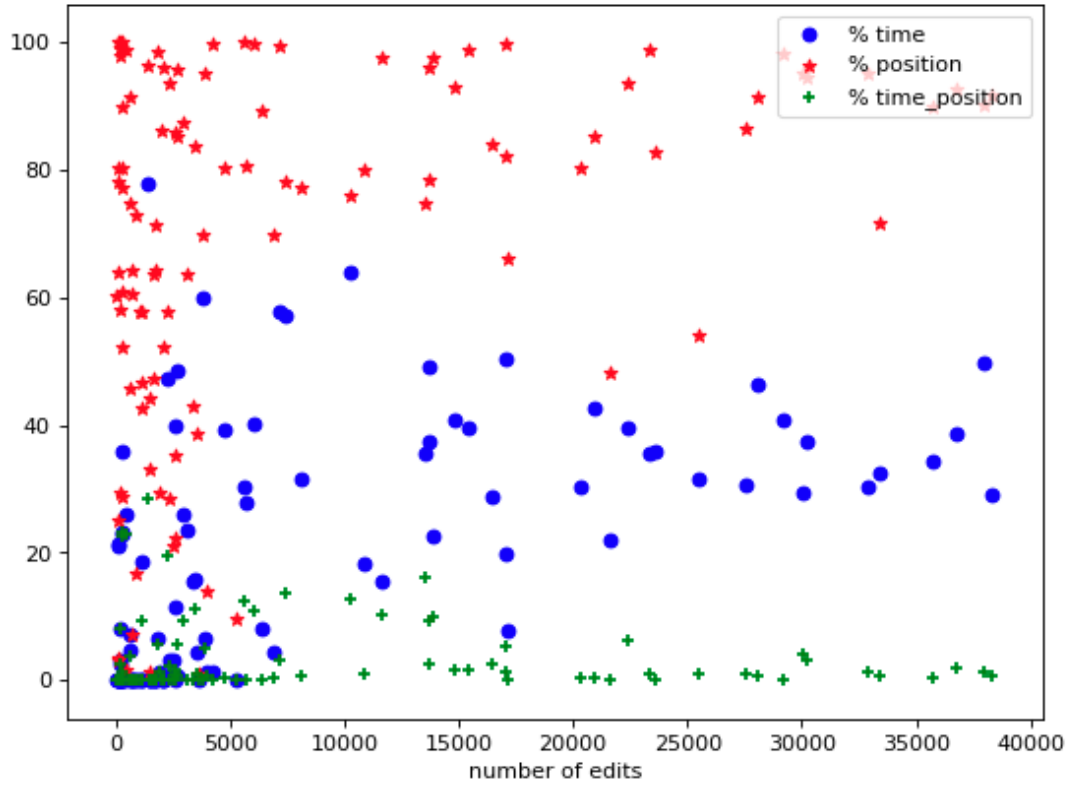


FIGURE A.1 – Proportion of time, position and time-position collaborative edits over the total number of edits of each document - [10 seconds, 80 characters] window

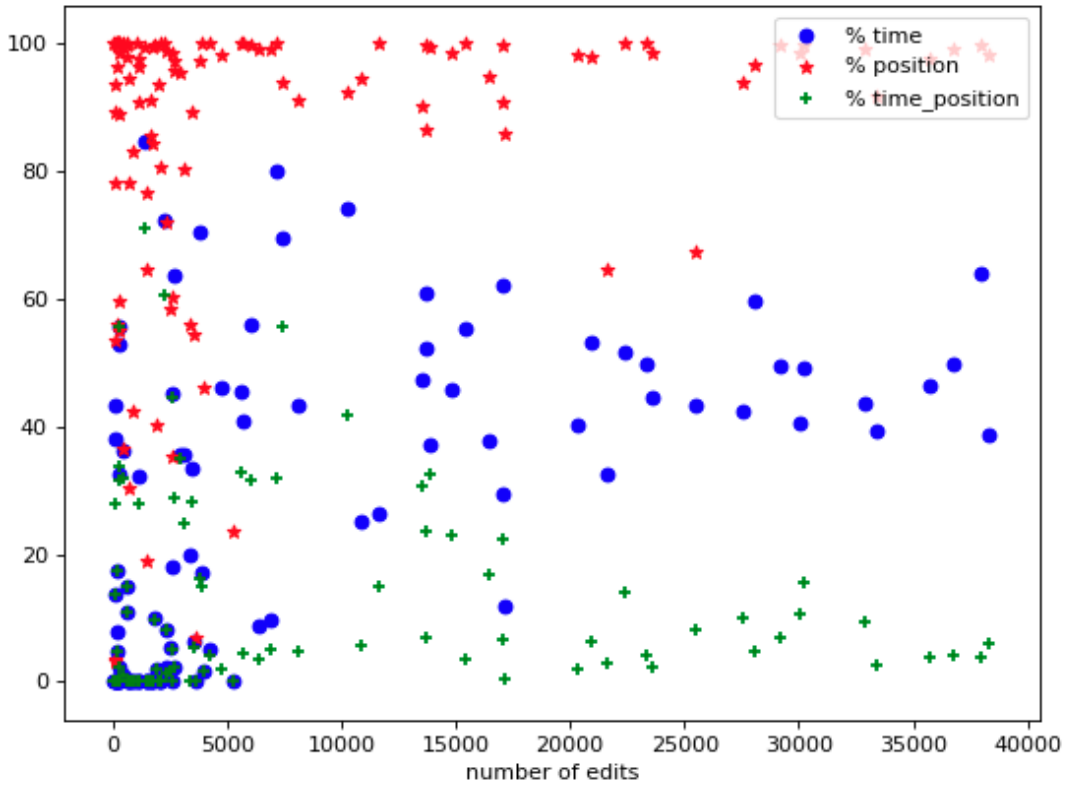


FIGURE A.2 – Proportion of time, position and time-position collaborative edits over the total number of edits of each document - [30 seconds, 400 characters] window

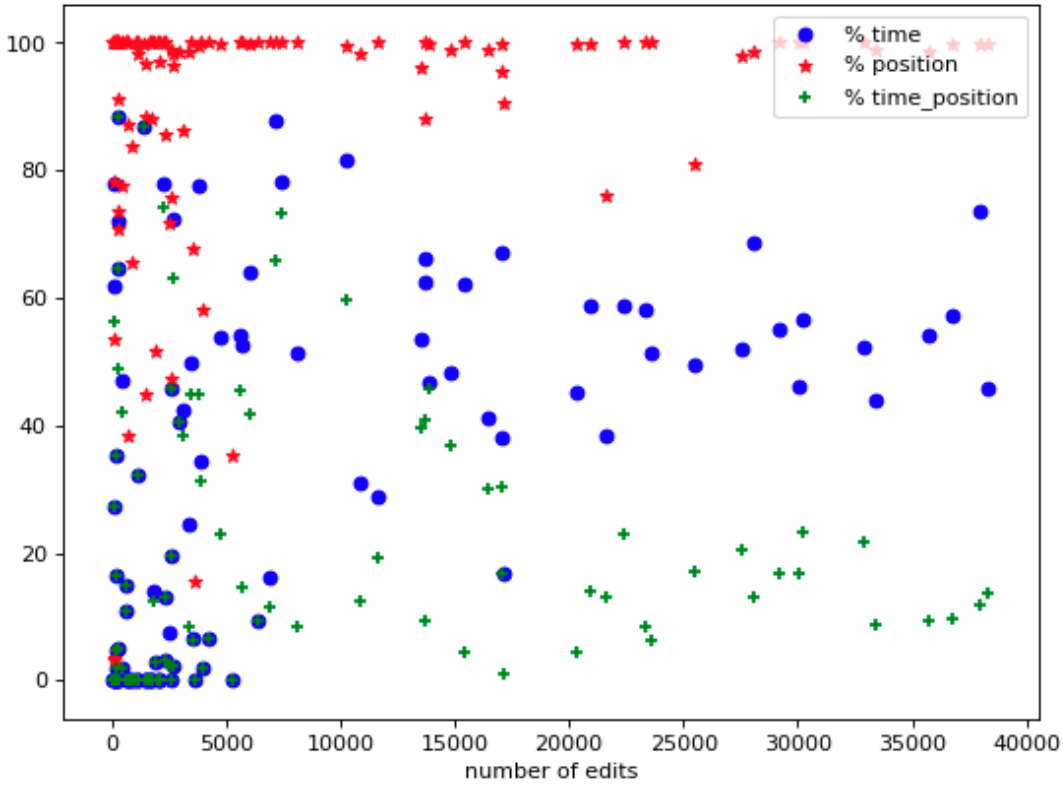


FIGURE A.3 – Proportion of time, position and time-position collaborative edits over the total number of edits of each document - [60 seconds, 800 characters] window

Appendix B

Clustering display of real documents

In this annex, we selected to present four documents that include both 'single-authored' sessions and 'co-authored' sessions and have different collaborative editing process.

id=59e8f8d98e96ef7e2dc01eb2 : This document has five 'single-authored' sessions and one 'co-authored' session. It was first created and edited by 'orange' author ('<' author). After four 'single-authored' sessions of the 'orange' author ('<' author), the 'green' author ('>' author) join the CE process and they had one 'co-authored' session which is around 1000 seconds. The document then was edited again by only the 'orange' author ('<' author).

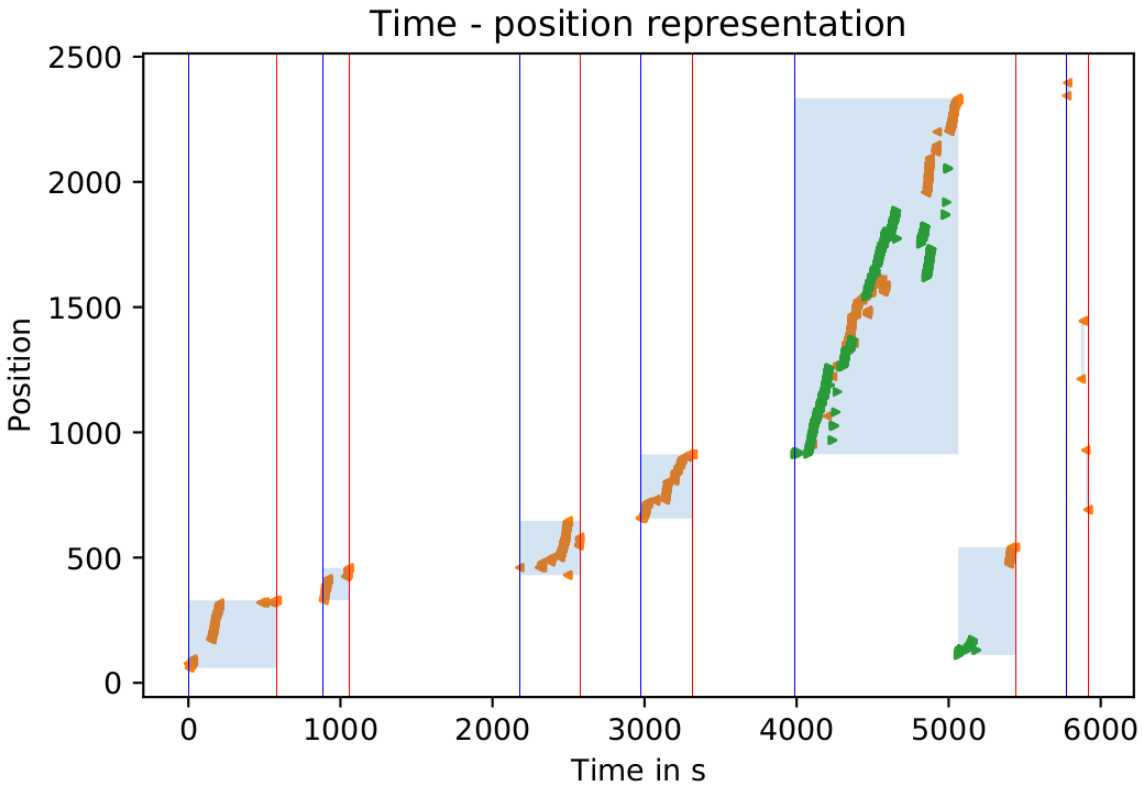


FIGURE B.1 – A real document(id=59e8f8d98e96ef7e2dc01eb2) presented in time-position view

id=59c918a874227c5d0cc75339 : This document was edited by four authors and has two 'co-authored' sessions. However, in the second 'co-authored' session, the 'orange' author ('<' author) have just one edit at the begin of the session and the remained edits belong to the 'green' author ('>' author). The first session was created and edited by the 'orange' author ('<' author). Then the 'green' ('>'), the 'red' ('V') and the 'violet' (square) authors joined to edit in collaboratively around 500 seconds. The rest of the session was edited by the 'green' author ('>' author).

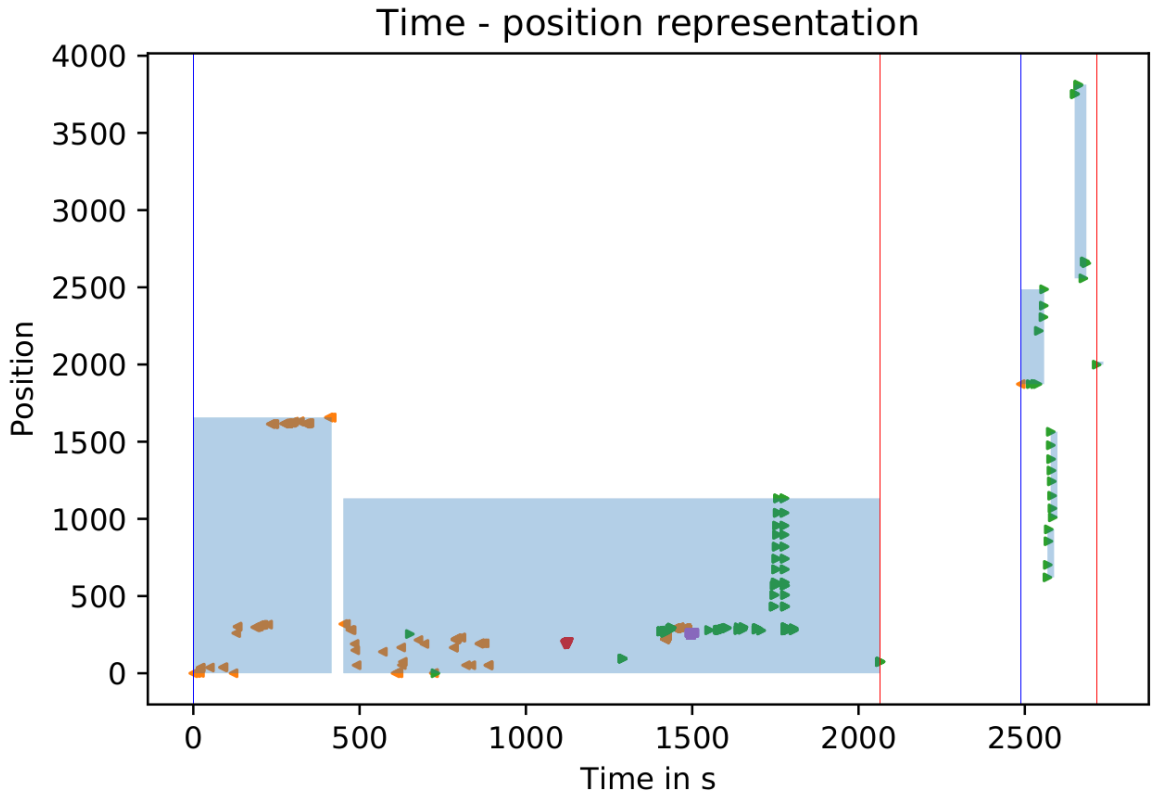


FIGURE B.2 – A real document(id=59c918a874227c5d0cc75339) presented in time-position view

id=59e91b788e96ef7e2dc01ee2 : This document has two 'co-authored' sessions. The first session was edited mainly by the 'green' author ('>' author). The two 'co-authors' edited together in the second session.

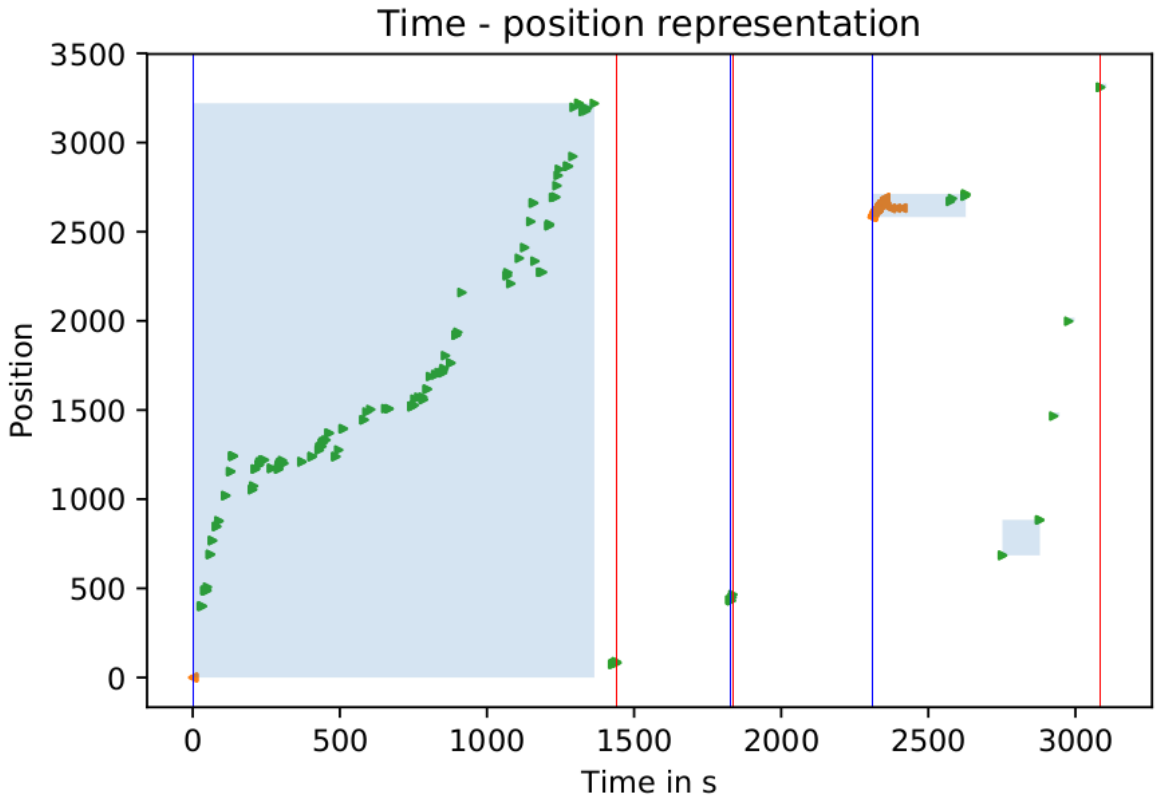


FIGURE B.3 – A real document(id=59e91b788e96ef7e2dc01ee2) presented in time-position view

id=59ede3ace7af18f16dbd7cc3 : This document has only one 'co-author' session in which the 'orange' author ('<' author) had edited somewhere and copied/pasted the content into ShareLaTex. They then edited together the content. It could be some 'minor' edits such as re-formatting text-styles, fixing typos.

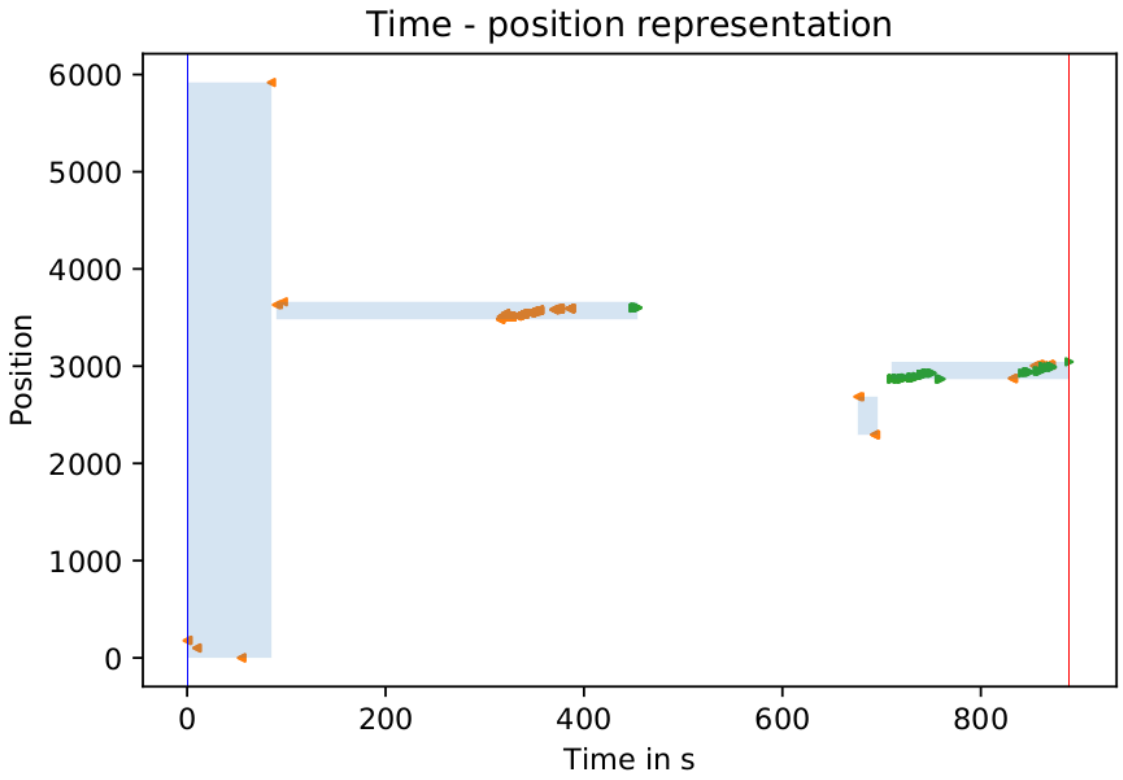


FIGURE B.4 – A real document(id=59ede3ace7af18f16dbd7cc3) presented in time-position view

Bibliography

[1] L. Ede and A. Lunsford. Singular texts/plural authors : perspectives on collaborative writing. Carbondale : Southern Illinois University Press, 1990.

[2] Carl Gutwin, Saul Greenberg, and Mark Roseman. Workspace awareness in real-time distributed groupware : Framework, widgets, and evaluation. In Proceedings of HCI on People and Computers XI, HCI '96, page 281–298, Berlin, Heidelberg, 1996. Springer-Verlag.

[3] I. R. Posner and R. M. Baecker. How people write together (groupware). In Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, volume iv, pages 127–138 vol.4, Jan 1992.

[4] Paul Benjamin Lowry, Aaron Curtis, and Michelle René Lowry. Building a taxonomy and nomenclature of collaborative writing to improve interdisciplinary research and practice. The Journal of Business Communication (1973), 41(1) :66–99, 2004.

[5] Sylvie Noël and Jean-Marc Robert. Empirical study on collaborative writing : What do co-authors do, use, and like ? Comput. Supported Coop. Work, 13(1) :63–89, January 2004.

[6] Sten Minör and Boris Magnusson. A model for semi-(a)synchronous collaborative editing. In Proceedings of the Third Conference on European Conference on Computer-Supported Cooperative Work, ECSCW'93, pages 219–231, Norwell, MA, USA, 1993. Kluwer Academic Publishers.

[7] Clarence A. Ellis, Simon J. Gibbs, and Gail Rein. Groupware : Some issues and experiences. Commun. ACM, 34(1) :39–58, January 1991.

[8] Julian Newman and Rhona Newman. Three Modes of Collaborative Authoring, pages 20–28. Springer, Dordrecht, 01 1992.

[9] Mike Sharples, J S. Goodlet, E E. Beck, C C. Wood, Steve Easterbrook, and L Polwman. Research issues in the study of computer supported collaborative writing. Computer Supported Cooperative Work, pages 9–28, 01 1993.

[10] Hee-Cheol (Ezra) Kim and Kerstin Eklundh. Reviewing practices in collaborative writing. Computer Supported Cooperative Work, 10 :247–259, 06 2001.

[11] Paul Dourish. The parting of the ways : Divergence, data management and collaborative work. In Proceedings of the Fourth Conference on European Conference on Computer-Supported Cooperative Work, ECSCW'95, pages 215–230, Norwell, MA, USA, 1995. Kluwer Academic Publishers.

[12] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving file conflicts in the ficus file system. In In Proceedings of the Summer Usenix Conference (USTC'94), pages 183–195, 1994.

[13] Dewayne E. Perry, Harvey P. Siy, and Lawrence G. Votta. Parallel changes in large-scale software development : an observational case study. ACM Transactions on Software Engineering and Methodology, 10(3) :308–337, 2001.

[14] Thomas Zimmermann. Mining workspace updates in cvs. In Proceedings of the Fourth International Workshop on Mining Software Repositories, MSR '07, pages 11–, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Early detection of collaboration conflicts and risks. IEEE Transactions on Software Engineering, 39(10) :1358–1375, Oct 2013.

[16] Bakhtiar Khan Kasi and Anita Sarma. Cassandra : Proactive conflict minimization through optimized task scheduling. In Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pages 732–741, Piscataway, NJ, USA, 2013. IEEE Press.

[17] Git. Fast version control system, 2005. http ://git.or.cz/.

[18] Brian Berliner. CVS II : Parallelizing Software Development. In Proceedings of the USENIX Winter Technical Conference, pages 341–352, Washington, D. C., USA, January 1990.

[19] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. Version control with Subversion. O'Reilly & Associates, Inc., 2004.

[20] T. Mens. A state-of-the-art survey on software merging. IEEE Trans. Softw. Eng., 28(5) :449–462, May 2002.

[21] Judith Olson, Gary Olson, Marianne Storrosten, and Mark Carter. Groupwork close up : A comparison of the group design process with and without a simple group editor. ACM Trans. Inf. Syst., 11 :321–348, 10 1993.

[22] GoogleDocs. Create and share your work online, 2006. http ://docs.google.com.

[23] ShareLaTeX. ShareLaTeX, Online LaTex editor., 2017. https ://www.sharelatex.com/. Accessed 19 October 2017.

[24] Etherpad. Open Source online editor providing collaborative editing in really real-time, 2018. https ://etherpad.org/.

[25] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, SIGMOD '89, page 399–407, New York, NY, USA, 1989. Association for Computing Machinery.

[26] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for p2p collaborative editing. In Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work, CSCW '06, pages 259–268, New York, NY, USA, 2006. ACM.

[27] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, Stabilization, Safety, and Security of Distributed Systems, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[28] Yunting Sun, Diane Lambert, Makoto Uchida, and Nicolas Remy. Collaboration in the cloud at google. In Proceedings of the 2014 ACM Conference on Web Science, WebSci '14, pages 239–240, New York, NY, USA, 2014. ACM.

[29] Judith S. Olson, Dakuo Wang, Gary M. Olson, and Jingwen Zhang. How people write together now : Beginning the investigation with advanced undergraduates in a project course. ACM Trans. Comput.-Hum. Interact., 24(1) :4 :1–4 :40, March 2017.

[30] Jeremy Birnholtz, Stephanie Steinhardt, and Antonella Pavese. Write here, write now ! : An experimental study of group maintenance in collaborative writing. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '13, pages 961–970, New York, NY, USA, 2013. ACM.

[31] Gabriele D'Angelo, Angelo Di Iorio, and Stefano Zacchiroli. Spacetime characterization of real-time collaborative editing. Proc. ACM Hum.-Comput. Interact., 2(CSCW) :41 :1–41 :19, November 2018.

[32] Ruby on rails (the popular mvc framework for ruby), 2015. http ://rubyonrails.org/.

[33] Ikiwiki, 2015. http ://ikiwiki.info/.

[34] Samba - opening windows to a wider world), 2015. http ://www.samba.org/.

[35] Linux kernel, 2015. http ://kernel.org/.

[36] Darcs. Distributed. Interactive. Smart, 2003. http ://darcs.net/.

[37] J. Grudin. Computer-supported cooperative work : history and focus. Computer, 27(5) :19–26, May 1994.

[38] P. Johnson-Lenz and T. Johnson-Lenz. Post-mechanistic groupware primitives : Rhythms, boundaries and containers. In Saul Greenberg, editor, Computer-supported Cooperative Work and Groupware, pages 271–294. Academic Press Ltd., London, UK, UK, 1991.

[39] Peter H. Carstensen and Kjeld Schmidt. Computer supported cooperative work : New challenges to systems design. In In K. Itoh (Ed.), Handbook of Human Factors, pages 619–636, 1999.

[40] R. E. Kraut, J. Galegher, and C. Egido. Relationships and tasks in scientific research collaboration. Human-Computer Interaction, 3(1) :31–58, 1987-1988.

[41] Claudia-Lavinia Ignat, Stavroula Papadopoulou, Gérald Oster, and Moira C. Norrie. Providing awareness in multi-synchronous collaboration without compromising privacy. In Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work, CSCW '08, pages 659–668, New York, NY, USA, 2008. ACM.

[42] Computer-supported cooperative work, 2019. https ://en.wikipedia.org/wiki/Computer-supported-cooperative-work.

[43] Git-wikipedia, 2019. https ://en.wikipedia.org/wiki/Git.

[44] Sha-1, 2005. https ://en.wikipedia.org/wiki/SHA-1.

[45] Git-internals, 2008. https ://github.com/pluralsight/git-internals-pdf.

[46] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pages 345–355, New York, NY, USA, 2014. ACM.

[47] Github. Web-based Git repository hosting service, 2008. https ://github.com/.

[48] Bitbucket. Code, Manage, Collaborate, 2008. https ://bitbucket.org/.

[49] Cleidson R. B. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. Sometimes you need to see through walls : A field study of application programming interfaces. In Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work, CSCW '04, pages 63–71, New York, NY, USA, 2004. ACM.

[50] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining git. In Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, MSR '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[51] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work, CSCW '96, page 288–297, New York, NY, USA, 1996. Association for Computing Machinery.

[52] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. ACM Trans. Comput.-Hum. Interact., 5(1) :63–108, March 1998.

[53] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology, UIST '95, page 111–120, New York, NY, USA, 1995. Association for Computing Machinery.

[54] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Proving Correctness of Transformation Functions in Real-Time Groupware. In 8th European Conference of Computer-supported Cooperative Work - ECSCW'03, page 18 p, Helsinki, Finland, 2003. Colloque avec actes et comité de lecture. internationale.

[55] Nicolas Vidot, Michèle Cart, Jean Ferrié, and Maher Suleiman. Copies convergence in a distributed real-time collaborative environment. In In Proceedings of the ACM Conference on Computer-Supported Cooperative Work, pages 171–180, 12 2000.

[56] Gérald Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems. In IEEE Conference on Collaborative Computing : CollaborateCom 2006, Collaborative Computing : Networking, Applications and Worksharing, 2006. CollaborateCom 2006, pages 1–10, Atlanta, Georgia, USA, November 2006. IEEE. http ://ieeexplore.ieee.org/.

[57] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems, ICDCS '09, page 395–403, USA, 2009. IEEE Computer Society.

[58] Matthieu Nicolas, VIctorien Elvinger, Gérald Oster, Claudia-Lavinia Ignat, and François Charoy. MUTE : A Peer-to-Peer Web-based Real-time Collaborative Editor. In ECSCW 2017 - 15th European Conference on Computer-Supported Cooperative Work, volume 1 of Proceedings of 15th European Conference on Computer-Supported Cooperative Work - Panels, Posters and Demos, pages 1–4, Sheffield, United Kingdom, August 2017. EUSSET.

[59] Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. How do centralized and distributed version control systems impact software changes ? In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pages 322–333, New York, NY, USA, 2014. ACM.

[60] S. McKee, N. Nelson, A. Sarma, and D. Dig. Software practitioner perspectives on merge conflicts and resolutions. In 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 467–478, Sep. 2017.

[61] Daniel M. German, Bram Adams, and Ahmed E. Hassan. Continuously mining distributed version control systems : an empirical study of how linux uses git. Empirical Software Engineering, 21(1) :260–299, 2015.

[62] Mehdi Ahmed-Nacer, Claudia-Lavinia Ignat, Gérald Oster, Hyun-Gul Roh, and Pascal Urso. Evaluating crdts for real-time document editing. In Proceedings of the 11th ACM Symposium on Document Engineering, DocEng '11, pages 103–112, New York, NY, USA, 2011. ACM.

[63] A. Nieminen. Real-time collaborative resolving of merge conflicts. In 8th International Conference on Collaborative Computing : Networking, Applications and Worksharing (CollaborateCom), pages 540–543, 2012.

[64] Georgios Gousios. The GHTorrent dataset and tool suite. In Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, pages 233–236, May 2013. Best data showcase paper award.

[65] Roberto Di Cosmo and Stefano Zacchiroli. Software heritage : Why and how to preserve software source code. In iPRES 2017 : 14th International Conference on Digital Preservation, Kyoto, Japan, 2017.

[66] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. The software heritage graph dataset : Public software development under one roof. In Proceedings of the 16th International Conference on Mining Software Repositories, MSR '19, pages 138–142. IEEE Press, 2019.

[67] Cleidson R. B. de Souza, David Redmiles, and Paul Dourish. "breaking the code", moving between private and public work in collaborative software development. In Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work, GROUP '03, pages 105–114, New York, NY, USA, 2003. ACM.

[68] Jeremy P. Birnholtz and Steven Ibara. Tracking changes in collaborative writing : Edits, visibility and group maintenance. In Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW, pages 809–818, 02 2012.

[69] Dakuo Wang, Judith S. Olson, Jingwen Zhang, Trung Nguyen, and Gary M. Olson. Docuviz : Visualizing collaborative writing. In Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI '15, pages 1865–1874, New York, NY, USA, 2015. ACM.

[70] Carl Gutwin and Saul Greenberg. Workspace awareness. In Workshop on Awareness in Collaborative Systems at the ACM Conference on Human Factors in Computing Systems (CHI'97), Atlanta, 1997. ACM Press.

[71] Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In Proceedings of the 1992 ACM Conference on Computer-supported Cooperative Work, CSCW '92, pages 107–114, New York, NY, USA, 1992. ACM.

[72] Inah Omoronyia, John Ferguson, Marc Roper, and Murray Wood. A review of awareness in distributed collaborative software engineering. Softw. Pract. Exper., 40(12) :1107–1133, November 2010.

[73] 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, pages 72–81, New York, NY, USA, 2004. ACM.

[74] James Tam and Saul Greenberg. A framework for asynchronous change awareness in collaboratively-constructed documents. In Gert-Jan de Vreede, Luis A. Guerrero, and Gabriela Marín Raventós, editors, Groupware : Design, Implementation, and Use, pages 67–83, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[75] Prasun Dewan and Rajesh Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In Richard Harper and Carl Gutwin, editors, ECSCW, pages 159–178. Springer, 2007.

[76] Claudia-Lavinia Ignat and Gérald Oster. Awareness of concurrent changes in distributed software development. In In : Meersman R., Tari Z. (eds) On the Move to Meaningful Internet Systems : OTM 2008. Lecture Notes in Computer Science, vol 5331, pages 456–464, 2008.

[77] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In Proceedings of the 19th ACM SIGSOFT Symposium and the

13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, pages 168–178, New York, NY, USA, 2011. ACM.

[78] Q. Dang and C. Ignat. Performance of real-time collaborative editors at large scale : User perspective. In 2016 IFIP Networking Conference (IFIP Networking) and Workshops, pages 548–553, 2016.

[79] Claudia-Lavinia Ignat, Gérald Oster, Olivia Fox, Valerie L. Shalin, and François Charoy. How do user groups cope with delay in real-time collaborative note taking. In Nina Boulus-Rødje, Gunnar Ellingsen, Tone Bratteteig, Margunn Aanestad, and Pernille Bjørn, editors, ECSCW 2015 : Proceedings of the 14th European Conference on Computer Supported Cooperative Work, 19-23 September 2015, Oslo, Norway, pages 223–242, Cham, 2015. Springer International Publishing.

[80] Python scripts for analyzing parallelism and conflicting changes in git, 2017. https ://github.com/coast-team/ParallelismAndConflictingChangesInGit.

[81] Spearman rank correlation coefficient, 2018. https ://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient.

[82] Practical meta-analysis effect size calculator, 2017. http ://campbellcollaboration.org/escalc/ html/EffectSizeCalculator-SMD10.php.

[83] Claudia-Lavinia Ignat and Gérald Oster. Awareness of Concurrent Changes in Distributed Software Development. In Proceedings of the International Conference on Cooperative Information Systems (CoopIS'08), pages 456–464, Monterrey, Mexico, November 2008.

[84] L. André, S. Martin, G. Oster, and C. Ignat. Supporting adaptable granularity of changes for massive-scale collaborative editing. In 9th IEEE International Conference on Collaborative Computing : Networking, Applications and Worksharing, pages 50–59, Oct 2013.

[85] Weihai Yu, Luc André, and Claudia-Lavinia Ignat. A crdt supporting selective undo for collaborative text editing. In Proceedings of the 15th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - Volume 9038, pages 193–206, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[86] Code together in real time with teletype for atom, 2017. https ://blog.atom.io/2017/11/15/code-together-in-real-time-with-teletype-for-atom.html.

[87] Linux kernel mailing list archive, 2017. https ://lkml.org/lkml.

[88] Merging with diff3, james coglan, 2017. //https ://blog.jcoglan.com/2017/05/08/merging-with-diff3/.

[89] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. A Formal Investigation of Diff3. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[90] Ricardo Olenewa, Gary M. Olson, Judith S. Olson, and Daniel M. Russell. Now that we can write simultaneously, how do we use that to our advantage ? Commun. ACM, 60(8) :36–43, July 2017.

[91] Ida Larsen-Ledet and Henrik Korsgaard. Territorial functioning in collaborative writing. Comput. Supported Coop. Work, 28(3–4) :391–433, June 2019.

[92] An api for applying incoming updates to documents in real-time), 2019. https ://github.com/overleaf/document-updater.

[93] Hoai Le Nguyen and Claudia-Lavinia Ignat. Parallelism and conflicting changes in Git version control systems. In IWCES'17 - The Fifteenth International Workshop on Collaborative Editing Systems , Portland, Oregon, United States, February 2017.

[94] Hoai Le Nguyen and Claudia-Lavinia Ignat. An Analysis of Merge Conflicts and Resolutions in Git-based Open Source Projects. Computer Supported Cooperative Work, 27 :741–765, June 2018.

[95] Hoai Le Nguyen and Claudia-Lavinia Ignat. Time-position characterization of conflicts : A case study of collaborative editing. In Alexander Nolte, Claudio Alvarez, Reiko Hishiyama, Irene-Angelica Chounta, María Jesús Rodríguez-Triana, and Tomoo Inoue, editors, Collaboration Technologies and Social Computing, pages 65–80, Cham, 2020. Springer International Publishing.

[96] Marco Biazzini and Benoit Baudry. "may the fork be with you" : Novel metrics to analyze collaboration on github. In "Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics (WETSoM 2014), Jun 2014, Hyderabad, India", 2014.

[97] Antoine Pietri, Guillaume Rousseau, and Stefano Zacchiroli. Forking without clicking : On how to identify software repository forks. In Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, page 277–287, New York, NY, USA, 2020. Association for Computing Machinery.

Résumé

L'édition collaborative (EC) a depuis longtemps attiré l'attention des chercheurs du Computer-supported-cooperative work (CSCW). Les premières recherches sur l'EC (dans les années 1990 et au début de 2000) se concentrent sur la description des différentes caractéristiques d'EC sur la base d'interviews de personnes qui avaient participé à certains projets d'EC. Certaines recherches récentes sur CE commencent à analyser les journaux des activités CE pour étudier comment les gens éditent ensemble avec le support des outils CE modernes tels que les systèmes de contrôle de version Git et Google Docs.

D'un point de vue général, le processus d'EC est la synchronisation continue de 'multiples, parallèles flux d'activités' de collaborateurs. Si la synchronisation a lieu moins souvent, par exemple le développement d'un projet logiciel basé sur le système de contrôle de version Git, il est considéré comme un mode de travail 'asynchronous'. Et si la synchronisation a lieu dans un petit intervalle, par exemple en éditant un document partagé dans ShareLaTex, il est considéré comme un mode de travail 'synchronous'. Plus la divergence est longue, plus le conflit est susceptible de se produire pendant la synchronisation. La résolution des conflits coûte cher, surtout après une longue période de divergence. Il est important de comprendre la fréquence des conflits et la manière dont les utilisateurs résolvent les conflits dans de vrais projets CE pour garantir de bonnes performances et une expérience utilisateur dans l'édition collaborative. Dans la première partie de cette thèse, nous empruntons les traces de collaboration de quatre grands projets open source dans le système de contrôle de version Git pour mener notre analyse. Nous analysons différents types de conflits textuels qui surviennent au cours du développement et comment les développeurs résolvent ces types de conflits. En particulier concernant les 'adjacent-line conflict', nous avons constaté que les utilisateurs les résolvent principalement en appliquant les modifications des deux sites. En outre, nous analysons également la fréquence à laquelle les utilisateurs utilisent le 'roll-back to previous version' pour résoudre les conflits de fusion.

Le processus de CE basé sur l'éditeur collaboratif en ligne est plus spécifique. Il peut être divisé en plusieurs *'sessions'* d'édition qui sont effectuées par un seul auteur ou plusieurs auteurs. Ils sont notés respectivement *'single-authored session'* et *'co-authored session'*. Ce processus de fragmentation nécessite un *'intervalle'* ou *'intervalle de temps maximal'* prédéfini qui n'est pas encore bien défini dans les études précédentes. Dans la deuxième partie de cette thèse, nous analysons les journaux des travaux CE d'un étudiant d'une école d'ingénieurs utilisant ShareLaTeX qui ont été collectés et anonymisés à des fins de confidentialité. En examinant différents 'maximum time gap' de 30 secondes à 15 minutes sur les journaux, nous avons constaté que nous pouvons déterminer un 'maximum time gap' approprié pour diviser les activités d'EC en sessions en évaluant la distribution de la *'external-distance'*. De plus, nous avons analysé les activités d'édition au sein de chaque *'co-author session'*. Nous empruntons une fenêtre de position temporelle de [30 secondes, 10 caractères] pour examiner ces cas de 'potential conflict'. Le résultat montre que les gens éditent rarement de près dans les deux positions temporelles. Cependant, les conflits sont plus susceptibles de se produire dans ces cas.

Mots-clés: édition collaborative, conflit, contrôle de version, éditeurs collaboratifs en temps réel

Abstract

Early researches about Collaborative Editing (CE) (in the 1990s and the early 2000) focused on describing different characteristics of CE based on interviewing people who had participated in some CE projects. Some recent researches about CE started analyzing the logs of CE activities to study how people edit together with support of modern CE tools such as Git version control systems and Google Docs.

From the general view point, the process of CE is the continuous synchronization of 'multiple, parallel streams of activity' of collaborators. If the synchronization takes place less often, for example in the development of a software project based on Git version control system, the work mode is called 'asynchronous'. And if the synchronization takes place within a small interval, for example in editing a shared document in ShareLaTex, the work mode is called 'synchronous'. The longer the divergence is, more conflicts are likely to happen during the synchronization. Resolving conflicts is costly, especially after a long period of divergence. Understanding how often conflicts happen and how users resolve conflict in real CE projects is important to ensure good performance and user experience in collaborative editing. In the first part of this thesis, we study the collaboration traces of four large open source projects in the Git version control system. We analyze different types of textual conflicts that arise during the development and how developers resolve these types of conflict. In particular regarding 'adjacent-lines conflicts', we found that users mostly resolve them by applying changes from both sites. Besides, we also analyze how often users use 'roll-back to previous version' as a way to resolve merge conflicts.

The process of CE based on an online collaborative editor is more specific. It can be split into several *'sessions'* of editing which are performed by a single author or several authors. They are denoted as *'single-authored session'* and *'co-authored session'* respectively. This fragmentation process requires a predefined *'interval'* or *'maximum time gap'* which is not yet well defined in previous studies. In the second part of this thesis, we analyze the logs of CE works of students of an Engineering School using ShareLaTeX which were collected and anonymized for privacy purposes. By examining different *'maximum time gap'* from 30 seconds to 15 minutes on the logs we found that we can define a suitable *'maximum time gap'* to split CE activities into sessions by evaluating the distribution of the *'external-distance'*. Besides, we analyse the editing activities inside each 'co-author session'. We borrow a [30 seconds, 10 characters] time-position window to examine these *'potential conflict'*cases. The result shows that people rarely edit closely in both time-position. However, conflicts are more likely to happen in these cases.

Keywords: collaborative editing, conflict, version control, real-time collaborative editors