Improving Textual Merge Result

Mehdi Ahmed-Nacer
Université de Lorraine
INRIA, LORIA
Email: mehdi.ahmed-nacer@loria.fr

Pascal Urso
Université de Lorraine
INRIA, LORIA
Email: pascal.urso@loria.fr

François Charoy
Université de Lorraine
INRIA, LORIA
Email: charoy@loria.fr

***Abstract*—In asynchronous collaborative systems, merging is an essential component. It allows to reconcile modifications made concurrently as well as managing software change through branching. The collaborative system is in charge to propose a merge result that includes user's modifications. The users now have to check and adapt this result. The adaptation should be as effort-less as possible, otherwise, the users may get frustrated and will quit the collaboration.**

The objective of this paper is to improve the result quality of the textual merge tool that constitutes the default merge tool of distributed version control systems. The basic idea is to study the behavior of the concurrent modifications during merge procedure. We identified when the existing merge techniques under-perform, and we propose solutions to improve the quality of the merge. We finally compare with the traditional merge tool through a large corpus of collaborative editing.

***Keywords*—Operational Transformation, collaborative editing, merging interfering, merge procedure, conflicts.**

I. Introduction

Collaborative editing systems allow multiple users distributed in time and space to edit the same shared document. To achieve high responsiveness and to support disconnected collaboration, data are optimistically replicated [34], [11]; i.e. each user has a local copy of the document that can be modified independently of the other replicas. In addition, to achieve high availability, locking mechanism to handle concurrent operations is prohibited. In peer to peer collaborative editing, the systems allow replicas to diverge temporarily, but must eventually reach the same value if no more mutations occur. This consistency model is called Eventual Consistency (EC) [41].

In asynchronous collaboration mode, e.g Distributed Version Control System (DVCS) softwares, users modify their document in isolation and synchronize after to establish a common view of the document. Usually, these kind of systems manages the modifications as a set of state (aka state-based approach) as on git system [39] or So6 [23]. When a replica receives a remote state, it computes the difference between the local state of the document and the received one before merging the modifications. If there are modifications in the same part of the document in both versions, the system can return a conflict information to the user and let him resolve them. The conflict is generated when the system cannot merge the concurrent modifications.

In order to provide a comfortable environment for collaboration, the collaborative editing system must merge correctly the modifications. Merging totally concurrent modifications on large scale collaboration is impossible. However, the system must reduce the human effort to obtain a correct merge. In the other case, the users correct by themselves the conflicts. If there is too much correction, the users may get frustrated and will quit the collaboration.

Many solutions have been proposed to improve automatic merge. A distinction can be made between textual [26], syntactic [14] and semantic [10] merging. Syntactic and semantic merge are more efficient than textual merge but they are specific to a given document. DVCS as git system supports any type of collaboration. The users can collaborate to produce XML files or a simple text document or software source code. For this purpose, in this paper we focused only of on textual merging.

Git system uses a $state - based$ approach to manage the concurrent modifications, called three-way-merge (diff3) [35], [16]. During the merge procedure, it compares the local state with the remote one. If the document is modified in the same position, diff3 produces a conflict. On the other hand, many $operation - based$ approaches were suggested to solve concurrency control in collaborative editing [9], [40], [1]. Unlike diff3, these approaches represent the modifications as a sequence of operations that are integrated automatically on the document. Both approach kinds are designed to reduce the effort of users during the collaboration. However, study what degree their result satisfy the users on real collaboration is never established. In this regard, this paper studies the behaviors of different approaches during the merge procedure and understand in which case they create conflicts to reduce the user's effort. This paper study for the first time a decentralized solution that offer a best merge than usual tool.

The contribution of this paper consists of observing through a tool, different patterns of collaboration in git histories. Afterward, we analyze the common cases that create a conflict during the merge procedure such as undo/redo operations [44], [31] and accidental clean merge [20]. Then, we adapt a solution to solve them by using operation-based approach.

We validate our contribution by several experiments on large scale histories. The experiments simulate traditional tool used for merging and the solution proposed. We measure the effort made by users in the document when a conflict is generated. Afterward, we compare our approach with traditional tool used for merging.

This paper is organized into seven sections. Section II describes the merge management by using existing approaches. Then, we describe our methodology and tool which allowed us to observe the different patterns of collaboration, detect

COLLABORATECOM 2013, October 20-23, Austin, United States
Copyright © 2013 ICST
DOI 10.4108/icst.collaboratecom.2013.254103

the different conflicts and compute the effort made by users during merging procedure. Section IV proposes a solution to correct the specific conflicts. Afterward, we present in section V the experimental evaluation of our approach and we analyze the performance of several existing collaborative editing algorithms. Finally, we cite the related work and we finish with a conclusion.

II. Merge Management

The merge result depends strongly on the type of algorithms used. In state-based systems as on git [39], the modifications are executed by states, while using operation-based algorithms the modifications are executed by operations. In the following, we describe how the modifications are managed in both approaches state-based and operation-based.

A. Diff3 tool

The usual tool used in asynchronous systems for synchronization and merging the documents is a three-way merges algorithm *diff3* [35], [16]. Developed in 1988 by Randy Smith and used in large version control systems such as CVS[7], Git [39] and SVN [6]. In Fig. 1, assume that the original document is O, user 1 modifies the document into A and user 2 modifies the document into B. When the collaborative system merges the document, diff3 finds the maximum matchings between O and A and between O and B. Then, diff3 examines the parts where O differs from either A or B and what has been changed by each user. Finally, it detects where the document conflict. Afterward, the system returns the results to the users with markers as in Fig. 1. These markers are useful, especially if the size of the document is large. It specifies exactly the position of conflict, in addition to other information, like the modifications made by other users and the original document. The users are invited to make corrections on their document to solve the conflict and add modifications if necessary.

B. Operational Transformation (OT)

Many operation-based algorithms are proposed and claim to integrate correctly the operations on the document. These algorithms respect the Eventual Consistency (EC) model; i.e, the systems allow replicas to diverge temporarily, but must eventually reach the same value if no more mutation occurs. In this paper, we assume that a granularity of operations is a line. So, the modifications are executed per lines.

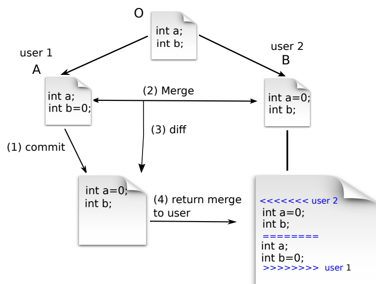


Figure. 1: Conflict in state-based systems for collaborative editing

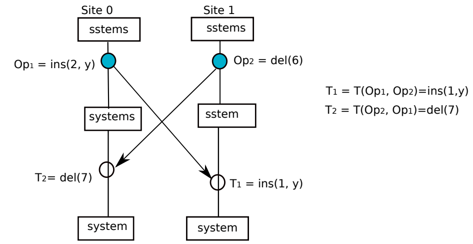


Figure. 2: Integrate operation in OT algorithms

Operational Transformation (OT) [9], [32], [23] algorithms are operation-based designed for collaborative editing context. They have been proposed to maintain the consistency of the shared document. For textual collaborative editing, they usually apply the $insert$ and $delete$ operations, and sometimes $update$ operations.
To apply the operations at the correct position and to preserve the user's intention, OT algorithms transform the operation received before its execution with the concurrent one, to take into account the changes made on the document by other executed operations. In Fig.2, two users shared the same document initially `"sstems"` and work together to produce the document `"system"`. User 0 inserts `"y"` at position 2 which intends to produce the document `"systems"`, when concurrently, user 1 deletes the character at position 6 which intends to produce the document ”sstem”. When user 0 receives op2, it is transformed to take into account the effect of the concurrent operation op1, then op2 is transformed to $del(7)$ instead of $del(6)$ since the position of the concurrent operation (op1) is before the position of op2. While, on site 1 the operation op1 is not transformed since the position of the concurrent operation (op2) is after the position of op1. Finally, both users produce the same document `"system"`.

Although, OT algorithms allow to order the operations, problems can happen when two users modify concurrently the text at *the same position* since there is no order between the operations.

However, deploying such algorithm on real system should not merge automatically every concurrent operation silently. It is more appropriate to inform the users and let him check the result. For example, so6 [23] that is similar to diff3 upon on OT algorithm, cannot merge silently the modifications when two concurrent operations are generated. the result is returned to the user and let him to solve the conflicts.

III. Methodology

In order to detect the merge behaviors that create conflicts, we deploy a framework[1] which allows us to observe the merge procedure and locate easily the conflicts. In addition, this tool replays the collaboration as on DVCS histories and computes the effort made by users on the conflicting document by using the traditional algorithms diff3 – state-based – and by using other operation-based algorithms. The difference is the gain in a user's effort.

However, to compute the effort made by users in case of conflict, we need to know what the users want as the final result

[1] https://github.com/score-team/replication-benchmarker

before starting the collaboration. The history of Distributed Version Control Systems (DVCS) contains the results that the user corrected. Thus, in the histories, the merge result is correct.

Assume that the modifications made by users to correct their document, when the conflict occur is the ground truth for merging procedure. The methodology consists of reproducing the same collaboration as on the histories of DVCS and observe through a tool the effort made by users when conflicts occur. At the first time by using the traditional algorithms (diff3), afterward by using our solution. Then, we compute the number of corrections performed to reproduce the same document as generated manually by the users. In textual merging [22], the most common approach is to use *line-based merging*. Thus, operation-based algorithms evaluated in this paper manage the modifications per lines. They create a new operation for each line modified.

A. Corpus available

A large number of available Distributed Version Control Systems (DVCS) history publicly available constitutes a very interesting corpus of distributed asynchronous editing traces. DVCS are widely used to manage large scale asynchronous collaborative editing. For instance, the linux kernel is developed by thousands of programmers around the world using Git [39]. Several web-based hosting services for software development projects provide large DVCS history such as GitHub (3.4M developers and 6.5M repositories)[2], Assembla (800,000 developers and more than 100,000 projects)[3], or SourceForge(3.4M developers and 324,000 projects)[4]. In this paper, we selected traces from the most used system: Git.

B. Framework

To replay the same collaboration as in the history of git by using operation-based algorithms, we need to transform the states to operations. Thus, we provide a framework which is the base of our experiment. The framework transforms the state of the document retrieved from the history of DVCS to the whole of operations ready to be used in operation-based algorithms. The framework implements also the operation-based algorithms and computes the size of modifications made by users to correct their document. The framework is open source and publicly available in order to let researchers evaluate their own algorithms. It is developed in Java, and reveal the source on GitHub platform[5] under the terms of the GPL license.

After retrieving the traces and implementing the framework, we replay the collaboration using operation-based algorithms and we compute the user's effort.

C. Merge computation

We define the user's effort as the difference between the simulated merge and the correct merge committed by the actual user. It represents the effort that users would make if the used DVCS system was based on the evaluated operation-based algorithms. We distinguish two metrics:

- **Merge blocks**: the number of different blocks on merged documents.
- **Merge lines**: the number of lines in the blocks or number of lines inserted by the framework to correct the document.

For example, if the user requires three consecutive insertions and two consecutive deletions to correct his document. Number of blocks are two and the number of lines is five.

Remark. *The framework does not count in merge metrics the markers add by the merge mechanism (lines beginning by “>>>>>>>>”, “<<<<<<<<” and “========”).the user need at lea s.*

D. Observing collaboration

In order to improve the merge procedure on asynchronous systems, we observe during the experiment through the framework, which part of document conflict and detect where the user's effort is the most important. This allowed us to understand the conflicts to solve them automatically.

The behavior of users during the collaboration is different from one project to another. Several factors can influence the collaboration such as, number of users, type of project, proximity between users, latency in networks ...etc. For this reason, it is difficult to detect and know what are the most common cases that create conflicts during the collaboration. The framework helps us to extract these scenarios.

1) Addition at the same position: The conflict happens when two users modify concurrently the text at the same position (not necessarily the same content) since there is no order between the operations.

In this kind of concurrency and on textual merge, operation-based algorithms can outperform state-based approach when an update operation falls in concurrency with insert operation as shown in Fig. 3. Initially, both users shared the same document `"int a;"`. afterward, user 1 updates the line by `"int a=0;"` when concurrently user 2 inserts in the same position `"int x;"`. After merging, git system cannot merge the documents since both users make modifications in the same position, contrary to the operation-based approach that merges the document correctly. Indeed, it transforms the update operation to a delete followed by an insertion, when user 2 receives the delete, it deletes `"int a;"` and after it inserts `"int a=0;"`, while user 1 applied the received insertion `"int x;"` in the correct position. Since there is no order between the concurrent operations, operation-based algorithm may also integrate the operations in the wrong order and force user to make corrections. Indeed, it is possible to produce the document ”`"int a=0;" "int x;"` instead of ”`"int x;" "int a=0;"` since there is not order between the concurrent operations. In this case, the user must delete `"int a=0;"` from position 1 and re-insert it at position 2. However, in the worst case, operation-based algorithm requires 2 modifications (one delete and one insert). While using state-based approach, the users require at least one modification to produce ”`"int x;" "int a=0;"` as for user 2 in Fig. 3, and in the worst case three modifications as for user 1. In this

[2]https://github.com/about/press
[3]https://www.assembla.com/about
[4]http://sourceforge.net/about
[5]http://github.com/PascalUrso/ReplicationBenchmark

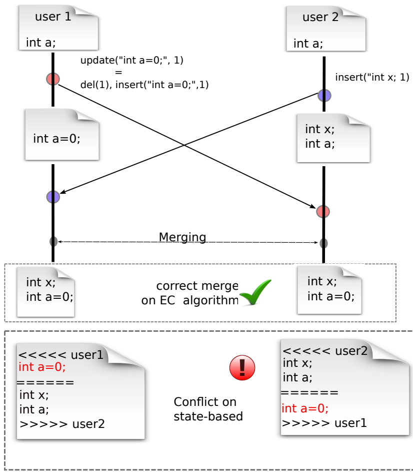


Figure. 3: Addition at the same position

example, the difference between the approaches is not very large (two modifications versus four modifications), but this difference is larger in real collaboration since users produce many copy/paste operations.
We notice also that this case of conflict is very common on real collaboration.

2) Accidental Clean Merge (ACM): When users insert the same content at the same position, this is called accidental clean merge. $diff3$ manages well this kind of conflict as in the Fig.4. Using operation-based algorithms that consider the modifications per line, a new operation is generated for each line, thus a duplicated line is inserted in the document and users must to correct line per line.

In Fig.4, both users insert concurrently the same element at the same position, `"int b;"` at position 2. $diff3$ detects that two lines are identical. Thus, diff3 merges correctly the document. While operation-based algorithm produces duplicated lines `"int b;"` `"int b;"` since it generates a different operation for each line.

3) Undo/Redo: The undo/redo operations are very useful on collaborative editing systems. They allow any user to correct any edit operation at any time. On git system the undo/redo operations are generated when users revert their modifications to one of the previous states[6]. However, using the operation-based algorithms can produce a conflict document.

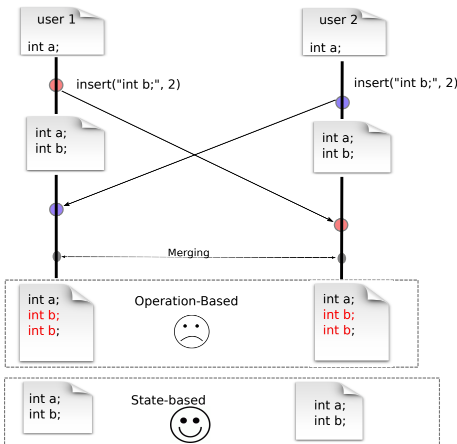


Figure. 4: Accidental clean merge

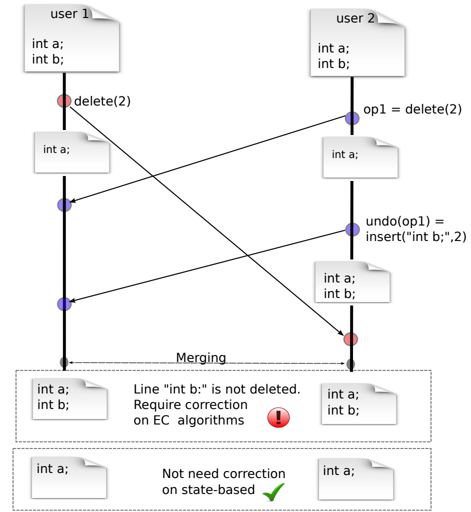


Figure. 5: undo/redo operation

Figure Fig.5 illustrates an example where state-based approaches manage well undo operations while operation-based algorithm creates a conflict. Initially sites 1 and 2 shared the same document `"int a;""int b;"`. Site 1 deletes line 2 which intends to produce the document `"int a;"`, while concurrently, site 2 deletes the same line and undo its operation and does not change the initial document. During the merge operation, $diff3$ of git system merges both states and produces a correct document `"int a;"`. While, operation-based algorithm creates a conflict in the user's documents. When site 1 receives the operations from site 2, it has reinserts `"int b;"` since site 2 undo its deletion. Thus both users produce `"int a;" "int b;"` document. To have the same document as in the history of git, both users must delete `"int b;"` from their document.

IV. Adapted Merge

To improve the performance of operation-based algorithms in asynchronous systems, we propose some improvement to avoid the "most" common cases that create conflicts: accidental clean merge and undo/redo conflicts. We adapt the Tombstone Transformation Functions (TTF) approach [25] to avoid these kind of conflicts. Before explaining our method we describe TTF algorithm.

A. TTF algorithm

TTF approach [25] was proposed to solve the problems occurred on Operational Transformation (OT) algorithms (described in section II-A). OT approaches are based on the transformation property C_1 and C_2 [32] and some transformation functions. C_1 ensure that the execution of any pair of concurrent operations obtains the same result on all replicas. Using a central server, C_1 is sufficient. However,

[6]Command "git revert"

on peer to peer collaboration the system require C_2 [42]. These functions change the index of the operation to take into account the effects of the concurrent operations. Imine et al. [13] have shown that few operational transformation algorithms proposed fail to satisfy C_1 and C_2 conditions. In this context, Tombstone Transformation Functions (TTF) approach was introduced [25]. It overcomes the problems by keeping all characters in the model of the document. When user deletes an element, it is not physically removed from the document, but just marked as invisible to users, i.e. deleted elements are replaced by tombstones. However, TTF approach does not solve the conflict described previously.

B. Clean Merge Undo Algorithm (CMUndo)

To improve TTF approach on asynchronous systems we add some transformation functions to take into account the case of undo/redo and accidental clean merge.

- **undo/redo**:
  Undo/redo operations in collaborative editing are very useful but considered as difficult problem [42], [44], [21], [5], [19]. They allow users to correct any edit operation at any time. In git system, the only information that can be useful to detect a real undo/redo operation, is the message introduced by users when they revert their modifications. Unfortunately, not all users specify on their messages that is a revert operation. For this reason, it is difficult to manage this kind of conflict by a revert mechanism. To simplify the operation, we assume that all $delete$ operations are considered as $undo$ of insert operation. Moreover, before inserting an element in the model we test if this operation is a $redo$ or a simple insert as shown in algorithm 1. The algorithm receives two arguments: position of insertion and the content of insertion. it returns the operation to be applied in the document and to be sent to other replicas. In line 1, the algorithm tests if it can find the element as a tombstone (invisible to users) in the same position. In this case this operation is considered as redo, in the other case it is considered as a simple insertion. [7]

Algorithm 1: LocalInsertion(pos, content)

```
  Input: The content and the position on the document
  Output: operation
1 if ((getDoc(pos).visibility = false) and
2 (getDoc(pos) == content) then
3   | return redo(position, content);
4 else
5   ⌊ return insert(position, content);
```

However, to manage the undo/redo operations, the algorithm uses the computation of *line visibility degree* [44]. When a line is created, it has a visibility of 1. Each time the line is deleted, the algorithm decreases its visibility degree. When a delete is undone or an insert is redone, the algorithm increases its line visibility degree. The line is visible only if its visibility degree is greater than 0.

[7]The user can delete an element, and after reinserts the same element in the same position without an explicit redo operation. During the collaboration there is a little chance to have this case.

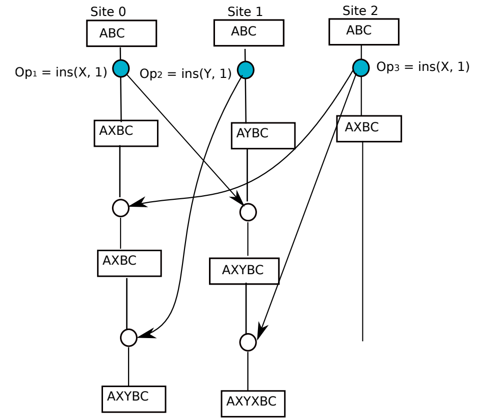


Figure. 6: ACM divergence by using traditional OT

- **Accidental Clean Merge (ACM)**:
  ACM [20] happens when users insert concurrently the same content at the same position. During the merge procedure, the merged document may contain a duplicated element. OT algorithms (described in section II-B) can be used to avoid these conflicts. They detect during the transformation phases the ACM cases and might transform them to $noop$ operations (nil value).
  To ensure consistency of the document when two concurrent operations made in the same position, TTF and other OT algorithms use site id as a priority [36], [30]. Using this solution with ACM transformation may create a divergence as presented in Fig. 6. Three sites shared the same document initially `"ABC"`. Site 0 and site 2 inserts concurrently the same element ”X” at position 1 and produce `"AXBC"` document. While, site 1 inserts concurrently ”Y” at position 2 and produces `"AYBC"` document. To avoid the ACM conflict, when site 0 receives the operation from site 2, it does not execute the operation since both users insert the same content at the same position. However, when site 0 receives the operation from site 1, it detects that both operations have the same position. Since OT algorithms give the priority to replica number, op2 is transformed with op1. It is transformed to insert at position 2 instead of position 1. Finally, site 0 produces `"AXYBC"` document. On the other hand, when site 1 receives op1 from site 0, it is not transformed, since the priority is given to site 0. Thus, site 1 produces `"AXYBC"` document. Afterward, when site 1 receives the operation from site 2, it transforms it to insert ”X” at position 3 and produces `"AXYXBC"` document. On site 1, ACM is not detected and replicas diverge.
  For this purpose, we propose a solution to use the element of operation as a priority. As an example, in this paper we chose the content's hash code. During the transformation, we add a new test to detect the accidental clean merge cases. Indeed, the algorithm 2 tests in lines 4 and 5 if there are two concurrent insertions in the same position with the same content. In this case, it returns a *noop* operation, in the other case it makes a traditional transformation by comparing the position and the content's code. Applying algorithm 2 in Fig. 6, the problem is resolved. Indeed, when site 1 receives an operation from site 2, the insertion

TABLE I: Projects characteristics

PROJECT	cloud/backbone	twitter/bootstrap	mbostock/d3	git/git	gitorious/mainline	jquery/jquery	rails/rails	statusnet/mainline
Head sha1	6ac7704c	37d0a30	d1d71e1	8c7a786b	c1105eb	2f2e045	36f7732	d7880c1
Files with merge	11	69	38	558	72	29	352	213
Commits	2293	6009	2192	32958	4136	5035	28895	12057
Merge	274	434	282	5646	151	178	1153	1218
Num.Operation	2605	7626	2352	33084	3915	5386	26899	11953
Max. Replica	13	10	30	59	5	12	6	11
Merge block Diff3	155	1614	648	3184	489	458	442	1159
Merge line Diff3	895	14658	4658	10159	2303	2146	3899	4783

of "X" is transformed into position 2 instead of position 3 since the hash code of "X" is less than hash code of "Y". Thus, the algorithm detects that two "X" are inserted in the same position. Site 1 detects ACM and does not execute op3. Finally, all replicas converge and produce "AXYBC" document.

Algorithm 2: Transform(op1, op2)

```
Input: operations to transform : op1 and p2
Output: operation applied on the document : op
1  Let c1 and c2 respectively the content of op1 and op2
2  Let t1 and t2 respectively the type of op1 and op2
3  Let p1 and p2 respectively the position of op1 and op2
4  if (t1 = insert) and (t2 = insert) then
5      if (c1=c2) and (p1=p2) then
6          return noop();/* An operation that
               return null value              */
7      else
8          if (p1 > p2) or (p1=p2 and
9          HashCode(c1) > HashCode(c2)) then
10             return insert(c1, p1+1,Site_i);
11         else
12             return insert(c1, p1,Site_i);
```

In the following, we provide an experiment to compare our solution with existing operation-based approaches, in addition to $diff3$ of git system.

V. Experimental Evaluation

The experiment was made on eight git repositories chosen among the most popular project from Github[8] hosting service and among the most active project from Gitorious[9].

A. Description of Collaboration Logs

Since there is no collaboration when the files are not merged, the framework replays only histories of files that are merged at least one.

In table I, we present the characteristics of eight projects. The head commit sha1 used to run our experiments is presented above the name of each repository. The characteristics are computed per file. Based on these files we compute the total number of commits and merges that affected the files, the number of operations and the maximum users that collaborate on each file of the project.

TABLE II: ACM and Undo/Redo in git repositories

Project \ Features	ACCIDENTAL CLEAN MERGE	UNDO	REDO
backbone	271	1357	1137
bootstrap	563	7210	3957
d3	7	19877	218
Git	1272	42734	1614
Gitorious	750	932	513
jquery	213	1947	1432
rails	426	5329	16172
status	2297	9060	6352

During the experiment, the framework computes the number of accidental clean merge and undo/redo cases. Table II presents the number of accidental clean merge and the number of undo/redo operations produced in the eight git repositories.

B. Algorithms Evaluated

Since git system is based on peer to peer architecture, the algorithms evaluated have the particularity that support peer to peer collaboration. We evaluated in this experiment, three-way-merge techniques (diff3 described in Section II-A) used by git system. We evaluated also the usual textual algorithm used for collaboration editing: Operational Transformation (OT) algorithms and Commutative Replicated Data Type (CRDT) algorithms [43], [33], [28], [24].
The most OT algorithms that exist, use a central component. However, some others do not require a central server such as SOCT2[37], MOT2[4] and Goto [38]. However, these algorithms require some property that only TTF approach ensures. In addition, the impact of these algorithms on merge result is same since they apply the same transformation functions. For this reason, we evaluated only SOCT2 among OT algorithms.

a) OT Algorithm: SOCT2: SOCT2 [37] algorithm is a representative Operational Transformation (OT) algorithm that do not make any assumption on using a central server for a total order of operations. The principle of this algorithm is illustrated in Fig. 7. When a causally ready operation is integrated on a site, the whole log of operations is traversed and reordered. After reordering, causally preceding operations come before concurrent ones in the history buffer. Finally, the remote operation has to be transformed according to the sequence of all concurrent operations.

b) CRDT Algorithm: Unlike OT algorithms, Commutative Replicated Data Type (CRDT) algorithms do not need to transformation functions. They are designed for concurrent operations to be natively commutative. In this paper we evaluate

[8]https://github.com/repositories.
[9]https://gitorious.org/projects.

the first CRDT proposed : WOOT [24]. In WOOT algorithm, the elements are uniquely identified. An insertion is defined by specifying the new element identifier, the element content and the identifiers of the preceding and following elements. Concurrent operations determine partial orders between elements. The merging mechanism can be seen as a linearisation of the partial order to obtain a total order.In Fig.8, two users shared the same document initially `ABC`. User 1 inserts `X` between `A` and `B` to produce `AXBC`, when concurrently user 2 deletes `B` and produces `AC`. The element deleted is just marked as invisible to users. When user 2 receives the operation from user 1, it is executed in a correct order. Since, each element has a unique identifier, when user 1 receives the operation from user 2, the correct element is deleted. However, if two concurrent insertions are generated in the same position, the merged operations can generate a conflict document.

WOOTH [1] is a new version of WOOT that improves its performance by using a hash table.

During the simulation of the collaboration, the framework computes number of corrections (Merge blocks and merge lines). Depending on the algorithms used and how an operation is generated, the order of blocks and lines in the document will be different. Thus, the number of corrections changes from one algorithm to another.

C. Result

Fig. 9a and Fig.9b represent respectively the percentage of merge blocks and merge lines for TTF, WOOT, diff3 algorithms and CMUndo. To observe how undo/redo and accidental clean merge operations impact on merge results, we present also Clean Merge (CM) algorithm that detects only accidental clean merge cases (without undo/redo operations). We consider the merge blocks and merge lines produced by diff3 of git system presented in table I as the reference (=100%).

The number of merge blocks and merge lines correlates well with the number of accidental clean merge and undo/redo operations represented in table II. Indeed, more accidental clean merge and undo/redo operations detected in repositories and more the difference between our solution and other algorithms grows. For example, in git repository we detected a large accidental clean merge and undo operations, so the gain of user's effort obtained by our solution is around 54% in git repository. In Gitorious repository, diff3 is more efficient

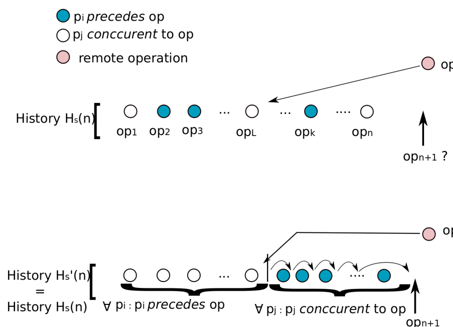


Figure. 7: Integrate a remote operation in SOCT2

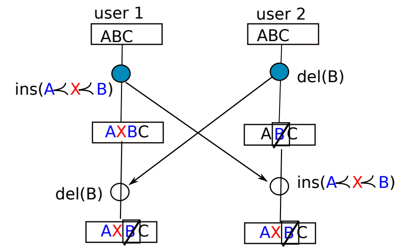


Figure. 8: Integration in WOOT

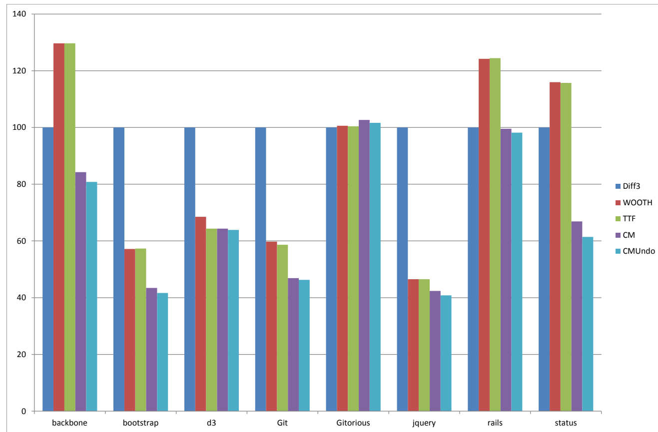
than all algorithms in merge block, This is due to a specific collaboration pattern in the file "`diff_browser.js`". The users collaborate independently and each one produces almost the same document. During the merge procedure diff3 manages well this kind of collaboration.

Even if OT and CRDT algorithms have a completely different behavior to merge the operations, the result of TTF and WOOT are almost the same in merge block and merge line. So, change the manner of operations' generation is not sufficient to improve the quality of the merge and reduce the users effort. However, CMUndo algorithm implements more functions to detect the accidental clean merge and undo/redo operations. It reduces in all cases the effort of users, except on Gitorious repository.

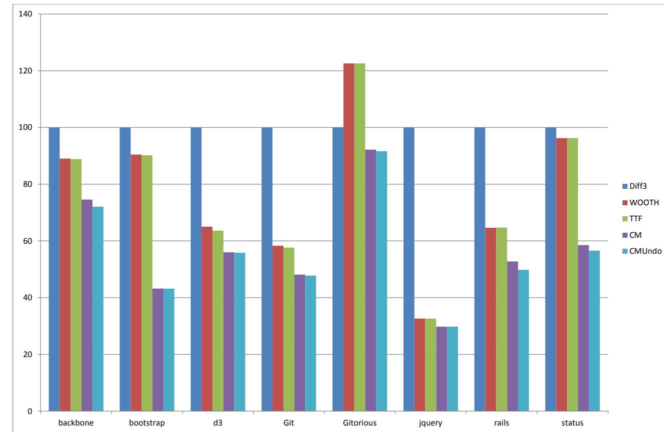
In Fig. 9a and on repositories that contain much accidental clean merge and undo/redo operations, diff3 algorithm outperforms TTF and WOOT algorithms but remains worse than CMUndo algorithm. Indeed, TTF and WOOT algorithms do not manage the accidental clean merge operations (see Fig. 4), while diff3 algorithm can merge them correctly and can retrieve some identical lines when two concurrent blocks are inserted. CMUndo is more efficient than all other algorithms except on Gitorious repository. Indeed, CMUndo takes the advantage of diff3 since it detects accidental clean merge operations and takes the advantage of operation-based algorithms since it manages well the concurrent addition at the same position. Except for Gitorious repository, CMUndo algorithm is the best.
In Gitorious repository diff3 is more efficient than all algorithms on merge block, This is due to a specific collaboration pattern on the file "`diff_browser.js`". The collaboration in these file begin with the merge of two branches that have no ancestor in common. However, these two branches contain states with common lines that the diff3 tool is able to merge.

However, the impact of accidental clean merge operations on merge result is greater than undo/redo operations. Indeed, CM algorithm that manages only accidental clean merge cases and CMUndo algorithm that manages accidental clean merge and undo/redo cases improve almost the same merge result. The difference is only 3%.

Using diff3 algorithm on asynchronous system creates more conflicts that CMUndo algorithm, Consequently, the document cannot be merged and users make more correction on their document. Comparing diff3 and CMUndo algorithms, the later gain 60% of blocks in jquery repository, 54% on git repository and 59% on bootstrap repository. However, it loses just 1% on Gitorious repository.



(a) Merge blocks on git projects



(b) Merge lines on git projects

Figure. 9: Merge blocks and merge lines

In Fig. 9b, it is clearly that CMUndo algorithm is the best. It outperforms widely all other algorithms and especially diff3 algorithm. More algorithm generates merge blocks and more the document require corrections. In Fig. 9a we found that CMUndo generate less blocks than diff3 algorithm, for this reason the users introduce many lines by using diff3 algorithm than CMUndo algorithm.
In addition, when a conflict occurs there is a high probability to generate a large block in state-based than operation-based approaches. Indeed, using operation-based approach, some operations can be inserted correctly, while on state-based approaches, the merge procedure depends on blocks. Then, when states are mixed the users require much correction. For this reason, TTF and WOOT algorithms outperform diff3 algorithm on merge lines. In Gitorious repository and precisely in "`diff_browser.js`' file, two users insert concurrently a large block with a content almost the same. During the merge procedure, diff3 can merge correctly the identical lines while operation-based approaches do not. for this reason, diff3 outperform TTF and WOOT algorithms. We notice that, this kind of collaboration is specific and rarely comes.
Using CMUndo algorithm on asynchronous system, the users require few corrections, while diff3 algorithm creates more conflict and require more corrections. Comparing diff3 and CMUndo algorithms, the later gain 70% of lines on jquery repository, 52% on git repository and 57% on bootstrap repository.

To summarize the experiment, we compute the total merge blocks and merge lines on all repositories. We found that for 1335 files, we compute 5799 accidental clean merge, 118409 undo/redo operations, *a gain of 3583 blocks and 21675 lines by using CMUndo algorithm*. Fig. 10 presents the total merge blocks and merge lines. In addition, we separate both algorithms (accidental clean merge –CM– and undo/redo) from our approach to observe the effect of each one on the result. TTF gains 26% in blocks and 23% on lines, while our solution gains 43% blocks and 50% lines. Moreover, accidental operations has the greatest effect on the document with a gain of 40% in blocks and 45% on lines. While undo/redo operations represent a gain of only 5% on blocks and lines.

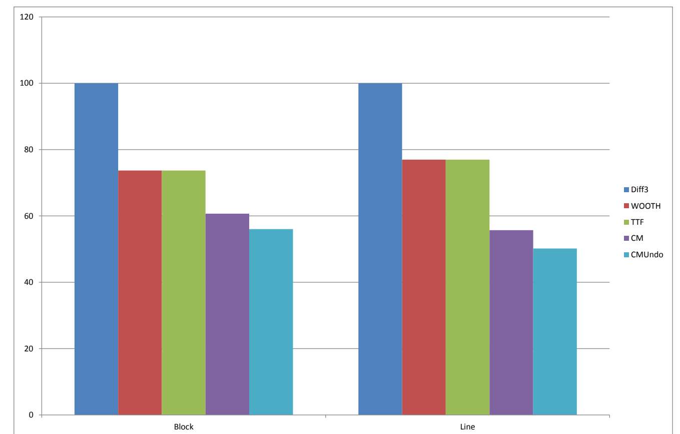


Figure. 10: Total merge block and merge line

VI. Related Work

Software merging is necessary during large scale development. It is very important to reduce the number of conflicts in the document during the collaboration. Textual, syntactic and semantic merging is widely studied in [22], [26], [14], [10], [15], [29]. However, git system deploys a generic model to allow any collaboration. The users can collaborate to produce XML files or a simple textual collaboration such as software source code. For this purpose, it is difficult to implement semantic and syntactic algorithms on git system. Thus, in this

paper we focused only on textual merging.

In [20], many policies are proposed to solve conflict in structured documents such as XML files or file systems. These policies can be applied with our methodology to manage the files of git system. As an example in this paper, we focused only in a simple linear text such as software development.

Bayou [27] proposed a technique to maintain the consistency of the shared document. It used an epidemic algorithm to propagate modifications between weakly consistent replicas. If the merge procedure cannot find a solution, conflict resolution is delegated to the user. However, the authors do not compute the conflicts and the efforts made by users. D.Perry et al. [26] studied the various aspects of parallel development in the context of a large scale software development. They observed a large collaboration and studied some interfering changes. However, they do not offer a solution to merge correctly the modifications. In [17], [26] the authors specify that 90% of the modifications can be merged without detection conflict and only 10% cannot be merged automatically, since the tool does not consider any syntactic or semantic information. The authors do not study the effort made by users to correct the conflicts. In [23], the authors propose an operational transformation algorithm that realizes a file system synchronization. However, The only operational transformation designed for collaborative editing and respect the transformation property C_1 and C_2 [32] is TTF approach evaluated in this paper.

On the other hand, operation-based algorithms designed for concurrency control such as Operation Transformation (OT) algorithms are widely studied on [9], [40], [1]. All the studies are focused on synchronous systems and they are focused on execution time or memory occupation. Recently new approaches called Commutative Replicated Data Type (CRDT) are proposed [43], [33], [28], [24] to be a substitution of OT algorithms. As OT algorithms, these approaches are evaluated only on execution time and memory occupation in [1] and [2]. Diff3 algorithm that is widely considered as the gold standard for merging document on asynchronous systems. It is widely studied and presented by many researchers in [16], [35], [18], [8]. However, study the merge result to reduce the user's effort in asynchronous system by using operation-based approaches are never studied.

An awarness mechanism can be independently added upon the same kind of merge algorithm without affecting their result [12], [3]. So, an awarness mechanism can be added in system upon CMUndo algorithm. If a conflict occurs, the system proposes to users an automatic merge and they can accept it without efforts. It is possible also to add modifications in the automatic merge if necessary.

In this regard, this paper studies for the first time a decentralized solution that can offer a better merge than usual tool.

VII. Conclusion

This paper presents an evaluation of eventual consistency algorithms in asynchronous systems, designed for collaborative editing. In addition, we present a solution to overcome the most cases of conflict that can be occur during the collaboration. We implemented also an open-source framework which allow us to observe the collaboration and detect the real conflict. The tool simulates a real collaboration as on the history of git repositories by using state-based and operation-based approaches. It computes the number of conflicts and the number of corrections requires by users to merge correctly their document.

Merging automatically the modifications can help users during the collaborations. When concurrent modifications occur, the merge tool can create conflicts. The users make an effort to correct their document. Reducing the user's effort improve the quality of collaboration and encourage users to work collaboratively.

In this paper, we observed the collaboration and studied the case where concurrent modifications interfere. We evaluated operation-based algorithm on asynchronous corpus. We found that, the existing operation-based algorithms perform well in asynchronous systems, but they do not manage any specific conflicts such as accidental clean merge and undo/redo operations. While diff3 algorithm handles these cases without problem.

For this purpose, we defined a new solution to avoid these kinds of conflicts and generate an operation-based algorithm that can be used correctly on asynchronous systems, reduce the conflicts and human interactions. It also outperforms the existing tool used on asynchronous systems: diff3.

Our experiments demonstrate in which cases operation-based algorithms are suitable for asynchronous systems and outperform the three-way-merge tool massively used in DVCS systems. We investigate first on the collaboration to detect the problems of merging procedure. Thus, we give guidelines to improve such OT algorithms to take into account the most common case that create conflicts when accidental clean merge and undo/redo operations are generated. Finally, we proposed a solution to handle these kinds of conflicts, make an experiment on asynchronous corpus, improve the quality of the merge and reduce the user's effort.

Acknowledgment

This work is partially supported by the ANR project Concordant ANR-10-BLAN 0208. The authors would like to thanks following people for their contributions to the algorithms implementation : G. Oster (SOCT2) and S. Martin (Git Walker).

References

[1] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso. Evaluating crdts for real-time document editing. In ACM, editor, *ACM Symposium on Document Engineering*, page 10 pages, San Francisco, CA, USA, september 2011.

[2] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, and P. Urso. 8émes journées francophones mobilité et ubiquité. In ACM, editor, *ACM Symposium on Document Engineering*, page 12 pages, IUT de Bayonne – Pays Basque, FR, jun 2012.

[3] S. Alshattnawi, G. Canals, and P. Molli. Concurrency awareness in a p2p wiki system. In *Collaborative Technologies and Systems, 2008. CTS 2008. International Symposium on*, pages 285–294, 2008.

[4] M. Cart and J. Ferrie. Asynchronous reconciliation based on operational transformation for P2P collaborative environments. In *Proceedings of the 2007 International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 127–138. IEEE Computer Society, 2007.

[5] R. Choudhary and P. Dewan. A general multi-user undo/redo model. In *ECSCW'95: Proceedings of the fourth conference on European Conference on Computer-Supported Cooperative Work*, pages 231–246, Norwell, MA, USA, 1995. Kluwer Academic Publishers.

[6] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion*. O'Reilly Media, 2007.

[7] B. Collins-Sussman, M. Pilato, and B. Fitzpatrick. Version control with subversion. 2003.

[8] D. M. P. Eggert and R. Stallman. *Comparing and Merging Files with GNU diff and patch*. Network Theory Ltd, January 2003.

[9] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. *SIGMOD Record : Proceedings of the ACM SIGMOD Conference on the Management of Data - SIGMOD '89*, 18(2):399–407, May 1989.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. 1999.

[11] O. A.-H. Hassan and L. Ramaswamy. Message replication in unstructured peer-to-peer network. In *CollaborateCom*, pages 337–344, 2007.

[12] C.-L. Ignat, S. Papadopoulou, G. Oster, and M. C. Norrie. Providing awareness in multi-synchronous collaboration without compromising privacy. In *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, pages 659–668. ACM, 2008.

[13] A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *Proceedings of the eighth conference on European Conference on Computer Supported Cooperative Work*, ECSCW'03, pages 277–293, Norwell, MA, USA, 2003. Kluwer Academic Publishers.

[14] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pages 243–252, Washington, DC, USA, 1994. IEEE Computer Society.

[15] A.-M. Kermarrec, A. I. T. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing - PODC'01*, pages 210–218. ACM Press, 2001.

[16] S. Khanna, K. Kunal, and B. C. Pierce. A formal investigation of diff3.

[17] D. B. Leblang. Configuration management. chapter The CM challenge: configuration management that works, pages 1–37. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[18] T. Lindholm. A three-way merge for xml documents. In *Proceedings of the 2004 ACM symposium on Document engineering*, DocEng '04, pages 1–10, New York, NY, USA, 2004. ACM.

[19] J. Maeda. *The laws of simplicity*. MIT Press, 2006.

[20] S. Martin, M. Ahmed-Nacer, and P. Urso. Controlled conflict resolution for replicated document. In *CollaborateCom*, pages 471–480, 2012.

[21] S. Martin, P. Urso, and S. Weiss. Scalable xml collaborative editing with undo. In R. Meersman, T. Dillon, and P. Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2010*, volume 6426 of *Lecture Notes in Computer Science*, pages 507–514. Springer, 2010.

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

[23] P. Molli, G. Oster, H. Skaf-Molli, and A. Imine. Using the transformational approach to build a safe and generic data synchronizer. In *Proceedings of the ACM SIGGROUP Conference on Supporting Group Work - GROUP 2003*, pages 212–220, Sanibel Island, Florida, USA, November 2003. ACM Press.

[24] G. Oster, P. Urso, P. Molli, and A. Imine. Data Consistency for P2P Collaborative Editing. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, pages 259–267, Banff, AB, Canada, November 2006. ACM Press.

[25] G. Oster, P. Urso, P. Molli, and A. Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *The Second International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2006)*, Atlanta, Georgia, USA, November 2006. IEEE Press.

[26] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, July 2001.

[27] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM symposium on Operating systems principles - SOSP'97*, pages 288–301. ACM Press, 1997.

[28] N. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia. A Commutative Replicated Data Type for Cooperative Editing. In *Proceedings of the 29th International Conference on Distributed Computing Systems - ICDCS 2009*, pages 395–403, Montreal, QC, Canada, June 2009. IEEE Computer Society.

[29] N. M. Preguiça, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003*, volume 2888 of *Lecture Notes in Computer Science*, pages 38–55. Springer, November 2003.

[30] M. Ressel and R. Gunzenhäuser. Reducing the problems of group undo. In *GROUP '99: Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 131–139, New York, NY, USA, 1999. ACM.

[31] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW '96*, pages 288–297, Boston, MA, USA, November 1996. ACM Press.

[32] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *CSCW*, pages 288–297, 1996.

[33] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354 – 368, 2011.

[34] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[35] R. Smith. distributed with gnu diffutils package, GNU diff3 (1988) Version 2.8.1, April 2002.

[36] M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work: the integration challenge*, GROUP '97, pages 435–445, New York, NY, USA, 1997. ACM.

[37] M. Suleiman, M. Cart, and J. Ferrié. Serialization of Concurrent Operations in a Distributed Collaborative Environment. In *Proceedings of the ACM SIGGROUP Conference on Supporting Group Work - GROUP '97*, pages 435–445, Phoenix, AZ, USA, November 1997. ACM Press.

[38] C. Sun and C. A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work - CSCW'98*, pages 59–68, New York, New York, États-Unis, November 1998. ACM Press.

[39] L. Torvalds. git, (April 2005). http://git-scm.com/.

[40] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, CSCW '00, pages 171–180, New York, NY, USA, 2000. ACM.

[41] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[42] S. Weiss, P. Urso, and P. Molli. An Undo Framework for P2P Collaborative Editing . In *CollaborateCom*, pages 529–544, Orlando, USA, November 2008.

[43] S. Weiss, P. Urso, and P. Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*, pages 404 –412, Montréal, Québec, Canada, jun. 2009. IEEE Computer Society.

[44] S. Weiss, P. Urso, and P. Molli. Logoot-undo: Distributed collaborative editing system on p2p networks. *IEEE Transactions on Parallel and Distributed Systems*, 21:1162–1174, 2010.