EAI Endorsed Transactions
on Collaborative Computing

Research Article **EAI.EU**

A Framework for Performance Evaluation of Decentralized Eventual Consistency Algorithms

Mehdi Ahmed-Nacer[1,2,*], Pascal Urso[2,†]

[1]Computer Science Department. University of Sciences and Technology Houari Boumediene, Algiers, Algeria.
[2]Université de Lorraine. INRIA, LORIA.

Abstract

Eventual Consistency (EC) model is adopted by numerous large-scale distributed systems. To ensure performance and scalability, this model allows any replica to accept updates without remote synchronization. Nowadays, many EC algorithms are developed to control the behavior of the replicated data in the face of concurrent updates. Among them, those using a central server to order the updates, while others support the decentralization. In this paper, we focus on decentralized EC algorithms. Suitability of such algorithms under users and devices constraints such as execution time, memory requirements, messages size and quality of the result remains to be investigated under different conditions. Evaluate such algorithms in different context and under different parameters require a framework.

In this paper, we propose a generic framework designed to evaluate different decentralized EC algorithms, in different context by controlling different parameters. Our framework provides a generic simulator that generates a runnable data following different parameters.

Keywords: Distributed Systems, Eventual Consistency, Operational Transformation, Commutative Replicated Data Types, Collaboration, Benchmark, Performance Analysis, Framework, Data Replication.

DOI: 10.4108/eai.30-6-2017.152755

1. Introduction

Replication is a key feature in any large distributed systems. When the replicated data are mutable, the consistency between the replicas must be ensured. A different model of consistency can be established. In the *strong consistency* model (aka atomic or linear consistency), a mutation seems to occur instantaneously on all replicas. Within a strong consistency model, integrity constraint on the data manipulated can be ensured through transactions. However, the CAP theorem [6, 10] states that it is impossible to achieve simultaneously strong consistency (C), availability (A) and to tolerate network partition (P).

In *Eventual Consistency (EC)* model, the replicas are allowed to diverge, but must eventually reach the same value if no more mutations occur. Eventual consistency promises better availability, performance and can be obtained in large-scale systems [29, 41]. To handle the structured eventual consistent data types, many mechanisms were proposed [1, 15, 31, 39] to control the behavior of the replicated data in the face of concurrent updates. Set [31], Text[1, 24, 45] and Tree [19] are different data types used in different context. They guarantees the consistency of the shared data following EC model. For instance, with a replicated structured document, adding concurrently two titles conflicts if the document type accepts only one title. To obtain a consistent document, *set* data type can be used. In collaborative editing systems, the effect of the concurrent modification must be the same for each user. Operational Transformation (OT) [8, 12, 27] and Commutative Replicated Data types (CRDTs) [24, 31] were proposed as the *text* data type to maintain the consistency of the document during the concurrent editing. However, *tree* data type are widely used to manage the concurrent updates for structured or semi-structured documents such as XML files and distributed file systems.

For each data type, many algorithms were proposed and claim that satisfy the EC model and suitable for users devices [7, 17, 26, 30, 33, 35]. Among of them, some need a central server to order the updates, while the other supports decentralized architecture. CRDT [1, 7, 24, 26, 31, 45] approaches were proposed in order to ensure convergence without blocking client operations and without having to deal with consensus. OT approaches [8, 27] for a decentralized architecture have been proposed also. While, Oster et al. show that

*Email: mehdi.ahmed-nacer@loria.fr
†Email: pascal.urso@loria.fr

most of them are incorrect [12]. The most representative OT algorithms that do not make any assumption on using a central server for a total order broadcast of operations are SOCT2 [33] and GOTO [35] algorithms. In this paper, we focused on EC algorithms that support the decentralization.

Analyze the performances of such algorithms, study their behaviors and their suitability according to different applications is not easy. In addition, a theoretical evaluation is not sufficient sometimes to meet the requirements. For instance, in collaboration editing many algorithms support well the number of operations during the edition, while they lose in performance when the number of copy/paste is important [1]. Due to their different natures, one can expect a different performance level than others. Until now, there is no open-source framework that allows researchers to evaluate their EC algorithms and compare them with others. The choice of the most suitable approach remains open, and the suitability of the algorithm is a question of performance and result quality. Therefore, we require a general and objective framework for performance comparison and analysis that allow users to choose the suitable algorithms according to their needs.

In this regards, we seek to provide an open source experimental framework that provides different decentralized EC algorithms, of different data types (set, text and tree), and to propose an evaluation methodology to compare them and its results on a set of representative algorithms. We present a principled framework that runs different algorithms in the same experimental setting and measures their performances. The proposed framework allows to determine if the performances of a given approach are suitable for some application context. It allows also to detect which factor most affects each algorithm's performance in term of execution time, memory usage, messages size and quality of merge results. Thus, depending on these factors we help to select which algorithm is the most adapted to which situation.

Because we are proposing a generic framework for performance comparison and analysis, it is important to design a mechanism in the framework that generates the needed data. Indeed, the framework integrates a simulator that generates a simulated data by giving to users the complete control of the parameters. In addition, the framework integrates a mechanism to extract a real traces and re-play them by using different EC algorithms. The framework supports two kinds of data: synchronous and asynchronous data.

The framework allows to each data type algorithm to generate an operation on their own format. It ensures also the causality order [36] between the operations during the experiment.

All the elements of the approach – algorithms and simulator implementations, as well as collaboration traces – are open-source and publicly available.

This paper is structured as follows. We begin in Section 2 by presenting our open-source framework that allows to compare eventual consistency algorithms on real and simulated traces. Afterwards, we present the different approaches evaluated. We establish then, the theoretical space and time complexity of these algorithms. In Section 3, we conduct the experiments and then we analyze the experimental measurements of the representative algorithms performance. In sections 5 we discuss about related work and finally in Section 6, we provide concluding remarks and directions for future works.

2. The Framework

The framework is called ReplicationBenchmark. It is developed in Java and open sourced under the terms of the GPL license [1]. It runs the different algorithms in the same experimental conditions and it measures their performances. It allows to understand which experimental variables influence most the different algorithms performances.

The Framework provides common base classes for different real-time collaborative editing algorithms, such as document, vector clocks [4], operations and simulator that disseminates the operations. Each algorithm can be implemented by derived classes. The Framework asks the replicas of each algorithm to generate an *operation* in its own format based on the *trace operation*. In our Framework, the operations generated by users (one character or copy/paste) are represented by the *TraceOperation* class, while *Operation* represents the operation ready to be executed by the algorithms. Figure 1 shown an overview of classes used in the framework.

The *VectorClock* class is used by some algorithms (see next section) to detect the concurrency between the operations. In addition, this class is used by the simulator to disseminate the operations in the same order than generated by the users in case of real traces. *TraceOperation* class represents the operation generated by the user. It contains some information such as the type of operation(insert or delete), the number of replicas, the vector clock of the current operation and the content of the operation in case of insertion or the size of the deletion.

However, the framework provides an interface *Trace* as presented in Figure 2. Following the traces desired, this interface is implemented in order to generate simulated or real traces:

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

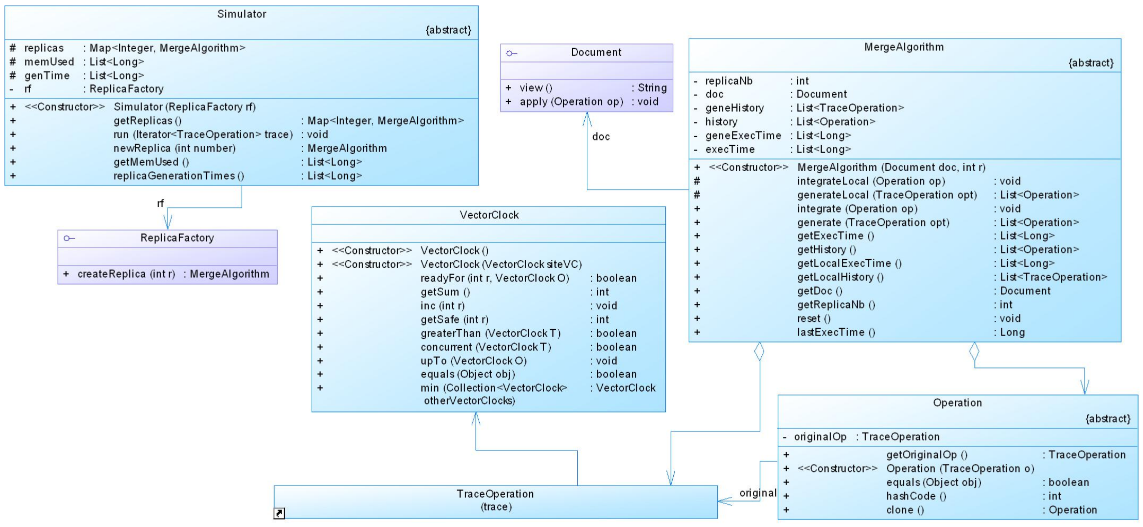


Figure 1. Framework core classes

1. Simulated traces: the framework can generate *linear* traces such as text used by textual data type algorithms, or *structured* traces such as XML traces used by tree data type algorithms. The *TextTrace* class allows to generate textual operations following the parameters specified in *OperationProfile* class. The latter class is controlled by the developers. Indeed, from this class, the developer can manage the traces by controlling many parameters such as the proportion of insertion, number of replicas, the proportion of copy/paste, etc. The different parameters that can influence the experiment, are controllable by the users. In Section 2.2, we detail more.

   XMLTrace class takes in the constructor, the XML document and an iterator from the root to traverse the tree. *XMLTrace* class generates a list of operations ready to be executed in tree data type algorithms.

2. Real traces: the framework generates also a real traces from a log – specified by the developer before the execution – or from DVCS histories. In [1], we made a real collaborative editing experiment with students. The operations made by users have been stored in a file. The *TraceFromFile* class extracts these operations from the file and allow the framework to reproduce the same collaboration as in real collaboration. The framework dispatches the operations in the same order as produced in the experiment.

   GitTrace class generates an operation from DVCS repositories such as Git. The framework traverses the histories of git repositories and simulate a collaboration composed of concurrent modifications. To simulate this collaboration, we adapted an open-sourced replication performance evaluation tool of distributed optimistic replication mechanisms [1].

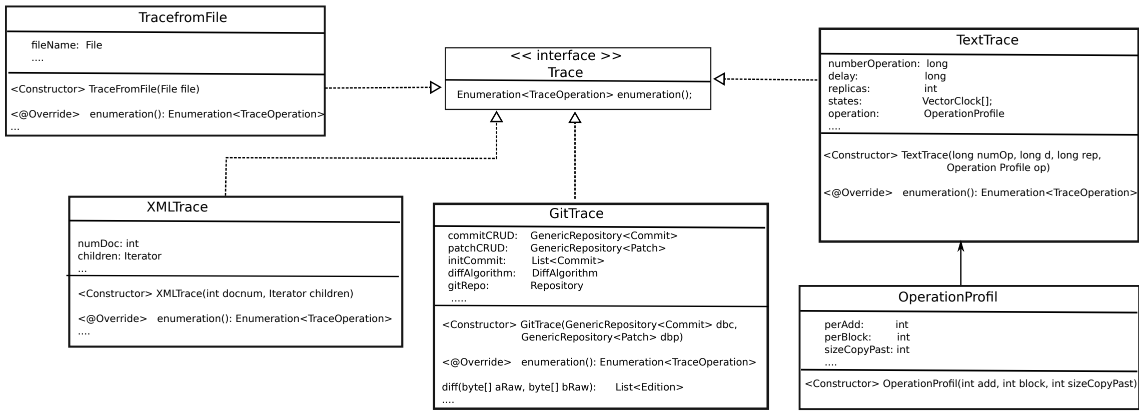


Figure 2. Framework trace classes

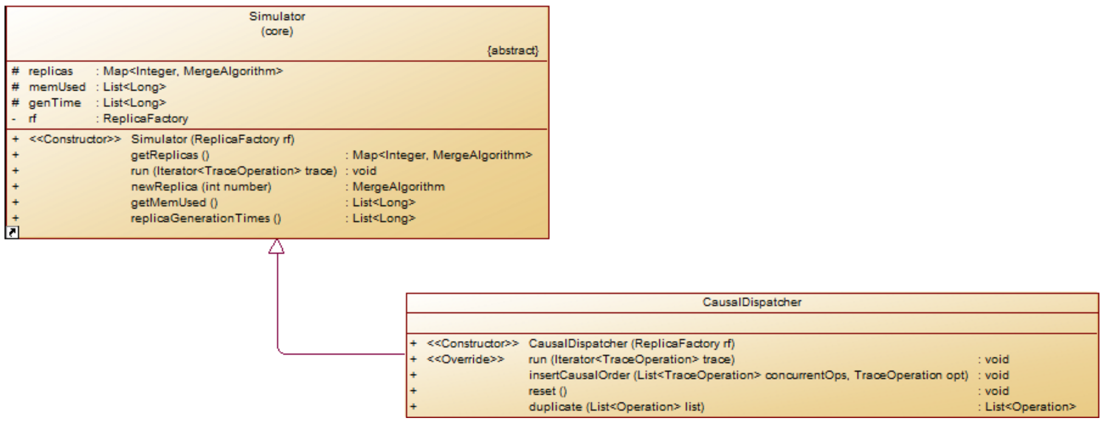

Simulator
(core)
{abstract}
# replicas : Map<Integer, MergeAlgorithm>
# memUsed : List<Long>
# genTime : List<Long>
- rf : ReplicaFactory
+ <<Constructor>> Simulator (ReplicaFactory rf)
+ getReplicas () : Map<Integer, MergeAlgorithm>
+ run (Iterator<TraceOperation> trace) : void
+ newReplica (int number) : MergeAlgorithm
+ getMemUsed () : List<Long>
+ replicaGenerationTimes () : List<Long>
CausalDispatcher
+ <<Constructor>> CausalDispatcher (ReplicaFactory rf)
+ <<Override>> run (Iterator<TraceOperation> trace) : void
+ insertCausalOrder (List<TraceOperation> concurrentOps, TraceOperation opt) : void
+ reset () : void
+ duplicate (List<Operation> list) : List<Operation>

Figure 3. Framework simulation classes

Operation and *Simulator* are abstract classes. Their implementation is in the derived classes. *Operation* class is implemented by each algorithm. It contains *trace operation* described previously, in addition to other methods of comparison. *Simulator* class is used to disseminate the operations. It contains the main method run implemented in *Causal Dispatcher* class as presented in Figure 3. *Causal Dispatcher* class, allows to each replica to generate a *trace operation* and dispatches the correspondent *operations* in correct order. Indeed, before disseminate or integrate the operations, run method calls insertCausalOrder to ensure the causality.

Document class is an interface with two methods: *view()* method to display the content of the document and *apply()* method to integrate a remote operation locally. *MergeAlgorithm* is an abstract class and extends by each algorithm. It initializes the document and it provides different methods to manage local and remote operations. For instance, *generate* procedure takes a *traces operation* in parameter and returns a list of *operations* in format of specific algorithms. In other words, it transforms the operation generated by the users to the list of operations ready to be executed by the algorithms. The implementation of this method is in the derived classes for generate a specific operation for each algorithm.

Finally, to measure the performances of different algorithms, we can summarize the framework operation through three fundamental phases:

1. Generate corpus: The framework gives to the users an entire choice of the corpus that will be used to evaluate the algorithms. Indeed, the users can choose between simulated or real data. In case of simulated data, the framework generates the operations following the parameters specified by the users such as: number of users, the proportion of insertions, number of operations, etc. In case of real data, the framework launches a mechanism to extract the operations from a log. In Section 2.2, we detail about the different data that can be generated by the framework. Whatever the mode used, the traces generated are a sequence of <Operation, replica, vector clock>, where the operation specifies the type of operations and their content, the replica is a unique identifier given to each user, while the vector clock is a structure used to ensure the causality [36] between the operations. The framework allows to each data type algorithm to generate an operation on their own format.

2. Specify format of operations and dissemination: The framework includes a dispatcher that disseminates the operations to the replicas following the order in the corpus. The operations are disseminated while respecting the causal order.

   According the order in the corpus and the replica that generates the operations, the framework allows the correspondent replica to generate a LOCAL OPERATION in their format. The replica executes the operation locally and the dispatcher broadcasts this operation to all other replicas. Each replica that receives this REMOTE OPERATION, it integrates it in their copies.

3. Compute the performances: To measure the response time of algorithms during the simulation, the framework stores the execution time of each local and remote operation using java.lang.System.nanoTime(). However, even using isolated systems, the garbage collector of java may affect the execution time. Thus, the framework is able to make several runs and to analyze the results in order to purge inconsistent values. To measure memory occupation and message size, the framework uses the default serialization interface of Java to estimate the memory footprint of a replica. The framework computes also the effort

made by users to correct their document after merges the updates.

To integrate a new algorithm in our framework, the developer has just to implement the different class described previously – Document, Operation and MergeAlgorithm–, in addition to the specific classes of the algorithm.

In the following section, we describe the different EC algorithms implemented and integrated into the framework.

2.1. Algorithms supported

Until now, the framework supports three different data types: Set, Text and Tree. For each data type, numerous algorithms that support the decentralization are implemented. However, the framework is implemented in generic way, it is easy to integrate another algorithm to the existing data type or integrate another data type such as Graph.

Set Data Type

. In our framework, the set ensures the eventual consistency by using Commutative Replicated Data Type (CRDT) approach [31]. A CRDT can be state-based or operation-based. In state-based CRDTs – aka Convergent Replicated Data Type (CvRDT) – the data are computed by merging the state of the local replica with the state of another replica. Eventual consistency is achieved if the merge relation is a monotonic semi-lattice. In the operation-based CRDTs – aka Commutative Replicated Data Type (CmRDT) – the data are computed by executing remote operations on the local replica. Eventual consistency is achieved if operations are delivered in a certain order and if the execution of the non-ordered operation commutes. For instance, using causal order, the execution of concurrent (in Lamport's definition [16]) operations must commutes. Our framework supports both cases: CvRDT and CmRDT. The different algorithms implemented are published an detailed in [5, 18, 31]. In the following, we give a brief description of each algorithm integrated into the framework:

LWW-Set In a Last Writer Wins Set (LWW-Set), each element is associated with a timestamp and a visibility flag. A local operation adds the element if not present and updates the timestamp and the visibility flag (true for *add*, false for *rmv*). The CvRDT merge mechanism makes the union of all elements and for each element the pair (timestamp, flag) of the maximum timestamp.

In the CmRDT, the execution of a remote operation updates the element only if the timestamp of the operation is higher than the timestamp associated with the element. The both CRDTs requires tombstones and the *lookup* returns elements which have a true visibility flag.

Counter-Set In this variation a counter is associated to each element. Let k be the value of the counter of an element. Initially 0 ,Adding an element increments the associated counter, and removing an element decrements it. In CmRDT A local *add* can occur only if $k \leq 0$ and sets the counter to 1 ($\delta = -k + 1$). A local *del* can occur only if $k > 0$ and sets the counter to 0 ($\delta = -k$). The CvRDT (also call PN-Set) payload contains the set of element, and for each element a set P of increment and a set N of decrement. A local *add*, resp. *del*, adds $|\delta|$ element in P, resp. N. The merge operation is the union of the sets. The lookup contains elements with $|P| > |N|$.

In the CmRDT, each operation contains the difference δ obtained during local execution. The remote operation execution adds δ to the counter. An element with a counter $k = 0$ can be removed, the others must be kept. The *lookup* contains elements with $k > 0$.

OR-Set In an Observed Remove Set (OR-Set) each element is associated with a set of unique tag. A local *add* creates a tag for the element and a local *rmv* removes all the tag of the element. The CvRDT contains the set of element, and for each element a set T of tags added and a set R of tags removed. The merge operation is the union of each set. The lookup contains elements with $T \cap R \neq \{\}$.

In the CmRDT, each operation contains the tag(s) added or removed. Since causal ordering is ensured and since tag are unique, the removed tag (and element with no tag) can be removed in the payload. The *lookup* contains the elements of the payload.

Optimized OR-Set In [5], the authors optimize the traditional OR-Set algorithm by minimizing the required meta-data. In *OptORSet*, each replica maintains a vector that indicates the observed n successive identifiers generated in others replicas. When a replica generates an operation, it increments its local counter. However, when the *add* is delivered, the element should have an effect only if it has not been previously delivered. Otherwise, it updates the counter of the remote replica and deletes the previous tags of the same element delivered from the same replica. *OptORSet* reduces the memory requirement by keeping only the last tag of the element peer replica.

Text Data Type

. For text data type, the framework supports two approaches that ensure the eventual consistency: Operational Transformation (OT) approach [8, 12, 27] and Commutative Replicated Data Types (CRDT) approach [24, 31].

Operational Transformation (OT) Algorithms: The Operational Transformation approach was introduced to overcome the problem of divergent copies in synchronous collaborative editors by serializing concurrent operations [8, 27, 34, 46]. It has been successfully employed in the massive context of GoogleDocs [13] which uses a centralized variation of the Jupiter algorithm [23]. The OT approach transforms a remote operation against its concurrent operation, i.e. ones that have changed at the same time the state of the object. It is based on the syntactic properties of the operations to preserve the user intentions. OT replicas store the history of all operations received. These histories are not necessarily the same, since the order of reception of concurrent operations is different for each replica. To ensure eventual consistency the transformation functions should satisfy some properties known as C_1 and C_2 [27]. By using a central server or a continuous global order [37, 40], the system only requires the condition C_1. Satisfying C_1 require to obtain the same result by executing in any order a pair concurrent operation defined on the same document state. In the general decentralized context [43], condition C_2 is required. C_2 expresses the equality between an operation transformed against two equivalent sequences of operations.

Conditions C_1 and C_2 ensure that transforming any operation with any two sequences of the same set of concurrent operations in different execution orders always yields the same result. Unfortunately, many proposed transformation functions fail to satisfy these conditions, as shown in [12]. To our best knowledge, the only existing transformation functions for collaborative editing that satisfy conditions C_1 and C_2 are the Tombstone Transformation Functions (TTF) [25]. To overcome problems, TTF approach keeps all characters in the model of the document, i.e. deleted characters are replaced by tombstones.

SOCT2 [34] is a representative decentralized operational transformation algorithm that requires the properties C_1 and C_2 to ensure convergence of replicas. As other decentralized algorithms [27, 35], it maintains a vector clock to ensure causality. When a user generates an operation, it is immediately executed locally, added to the local history buffer, and sent to all other replicas. The operation is broadcasted to other replicas with the identifier of the source site and its vector clock. The principle of this algorithm is that before a remote operation integration, the history of already executed operations is traversed and reordered. After reordering, causally preceding operations come before concurrent ones in the history buffer.

Traversing the history buffer is a costly operation, but mandatory to achieve correctness. The history buffer size can be reduced by removing entries seen by every replica, but such a garbage collection mechanism requires a consensus or a fixed and known number of replicas [14] which is not feasible in the general distributed context.

Commutative Replicated Data Types Algorithms: Commutative Replicated Data Types (CmRDT) are operation-based conflict-free replicated data types (CRDT) [31]. CRDTs ensure consistency of highly dynamic contents on peer-to-peer networks. Unlike OT algorithms, CRDTs require no history of operations, and no concurrency control to ensure consistency. Instead, CmRDT are designed for concurrent operations to be natively commutative by actively using the characteristics of abstract data types such as lists or ordered trees. However, CRDT algorithms have not yet been applied to massive collaborative editing in an industrial context.

WOOT [24] merge operations contain the element to be inserted with the preceding and the following element. In case where two elements have not a precedence relation between them, the priority is given according to their identifier. To ensure the consistency of the replicas, the deleted elements are not deleted but just marked as invisible to users.

WOOTO [42] improves WOOT complexity by using element degrees to compare unordered elements instead of the respective placement of the preceding and next elements. This optimization sightly improves computing time and model space but not message size since insertions operations still specifies the identifiers of the preceding and the next elements along with the degree.

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

RGA [28] specifies on the identifiers (aka *s4vector*) the last previous element visible during its generation. Thus, the tombstones also remain after the deletions.

Logoot [44] CRDT generates identifiers composed of a list of positions. The identifiers are ordered with a lexicographic order. A position is a 3-tuple containing a digit in a specific numeric base, a replica identifier and a clock value. Identifiers are unbounded to allow for arbitrary insertion between two consecutive elements. Unlike RGA and WOOT algorithms, Logoot does not need to store the tombstones since elements are not linked through insertion operations.[2]

LogootSplit Unlike Logoot that takes use a single elements granular, LogootSplit [3] algorithm identifies a continuous sequence generated by a user operation. Continuous sequences can be produced by real-time copy/paste, version patches, or operation buffering.

The major advantage of LogootSplit algorithm is the number of identifiers which is reduced, whereas the algorithm performance depends on the proportion of continuous sequences inserted.

Treedoc [26] is a CRDT algorithm that represents the document by a binary tree structure. The element identifier is the path to the element in the tree. If two users insert concurrently at the same position, Treedoc creates a major-node that contains the two elements.

Tree Data Type

. Trees are a fundamental data structure for many areas of computer science and system engineering. in addition, the eventual consistency is more difficult to achieve facing complex conflict resolution. Indeed, more the data type is complex, more conflicts appear. For instance, in a structured data such as XML files and file systems, modifications such as adding and removing an element, or adding a child node while removing the father to which it belongs, or setting concurrently two same elements conflict. Tree data types are useful to manage such conflicts [21]. Our framework provide a collection of tree algorithms based on OT and CRDT designed for hierarchical documents and semi-structured documents. In addition, the framework proposes a generic design to ensure eventual consistency and control the conflicts in such structures.

TreeOpt and OTTree TreeOPT (tree OPerational Transformation) [11] is a general algorithm designed for hierarchical documents and semi-structured documents. Each node contains an instance of an operation transformation algorithm [8, 27, 35]. The algorithm applies the operational transformation mechanism recursively over the different document levels. In our experimentation, we have used this algorithm with SOCT2 [33] algorithm and TTF (Tombstone Transformation Functions) approach [25]. For little optimization, we save only insertion operation in log of SOCT2 [34].

The OTTree, an unpublished algorithm, uses only one instance of SOCT2 for entire the tree (not on each node) and TTF on each child list. The operation of TTF and its integration function were modified to include the path information.

FCEdit FCEdit [20] is a CRDT designed for collaborative editing of semi-structured documents. It associates to each element a unique identifier. FCEdit maps $identifier \rightarrow node$. So it uses just a hash table to find an element in the tree. Each child is ordered by a position identifier. Unlike OTtree, FCEdit does not need to store an element in tombstone. The elements are really deleted from the tree making it more efficient in memory.

The framework proposes also a different policies to manage the concurrent modification for trees [19]:

1. Skip: drops the orphan path.
2. Reappear: recreates the path leading to the orphan path
3. Root: places the orphan subtree under the root
4. Compact: places the orphan subtree under its longest non-orphan prefix.

2.2. Obtaining corpus

To observe the behavior of different EC algorithms and study their performance, the framework needs to run these algorithms on a large corpus. The framework gives to users the entire choice of the evaluation: by using a realistic data or a simulated data. Indeed, the framework integrates a mechanism to extract a real data and a simulator that generates a simulated corpus following different parameters. In the following, we explain how the framework produces both corpus.

Real Traces

. The framework can produce two kinds of real corpus:

1. Synchronous editing corpus: There is no publicly available corpus of collaborative editing traces. The existing studies [17, 26, 45] were conducted on purely randomized and private traces or Wikipedia and/or SVN traces which are centralized and synchronous. Centralized traces do not contain concurrency between editions.

   In [1] we performed user studies on real-time peer-to-peer collaboration and we collected usage traces. We modified a real-time collaborative editor called TeamEdit [38] to log the user

[2]Since the Logoot algorithm generates its identifiers by using a random function and the order of these identifiers is not necessarily the same in two different executions, we conducted four executions and we computed the average metric.

operations generated during an experiment. For both operations insert/delete, we store a position, a version vector that represents concurrency, a replica identifier and a document identifier. In addition, for insert operations, we store the character or the block of characters inserted, while for delete operations, we store the number of characters deleted. We made two collaboration experiments with groups of students. The first collaboration was performed with three groups of four master students during two hours. We asked the students to write collaboratively a report without using any other communication tools except TeamEdit. In the second experiment, we asked two times eighteen students to watch two TvShow and to produce a transcription of the episode while watching it. In Table 1 we present some characteristics of the traces collected in [1].

Input data	Report	TvShow
Avg percentage of insertions	88.26%	93.23%
Avg number of operations	11993.00	9435.00
Avg number of block insertions	112.66	174.00
Avg number of block deletions	97.33	38.00
Avg number of replicas	4.00	18.00
Avg blocks size	158.89	34.45

Table 1. Characteristics of the collected synchronous traces

The framework can reproduce the same real traces by adjusting the features presented in Table 1.

2. Asynchronous editing corpus: A large quantity of distributed asynchronous collaborative editing data is publicly available. Several web-based hosting service for software development projects such as GitHub, Assembla, or SourceForge host numerous open-sourced software code bases. When this code is managed with a distributed version control system such as Git, Mercurial, or Bazaar, anyone got access to the history of concurrent editing. However, these histories must be treated to be run by other algorithms.

   The real concurrency information is not available in DVCs. The most used DVCS – git, Mercurial or Bazaar – do not manage replica information in their data storage. The only available information is the email of the user who produced a given commit and the merge history. The user's email is not reliable since it may be invalid, a same user may work on several replicas, or it may change its email while working on the same replica. The merge information is more reliable. A merge is a commit that has more than one parent. Different parents are different replicas or branches.[3] The inverse is not true since two consecutive commits without merge can be produced by different replicas.

   Our framework is able to parse git repositories to obtain large runnable traces of asynchronous collaborative editing. First, it begins by assigning a replica identifier to commit. It heuristically minimizes the number of replicas.

   A commit replica identifier is assigned to one of the parents of the commit. Thus, if two commits have the same replica identifier, there exists a path between them.

   Second, for each commit that has only one parent the framework computes the diff [22] between the two states. The diff result is a list of insert, delete or update operations concerning block of lines. For merge commits, the framework store the state of the resulting merge commit. The relation between the commits is represented by vector clocks. All commits (diffs and merge states) and their vector clocks, are stored in Apache CouchDB database [9] in order to be used by all the runs of the different algorithms.

   In DVCSs, merging is not a fully automated process. When a user merges, it obtains a best-effort result – computed using three-way-merge techniques. The user must change this result in case of conflicts. He may also change the result, if, for instance, the merged source code does not compile. The stored merge state is the result of the whole process. *Our framework simulate the same behavior during collaborative editing algorithm evaluation*. A best-effort merge is obtained using the evaluated algorithm and a "correction" is computed on-the-fly to obtain the result intended by the real user that done the merge. In addition, an algorithm that requires more corrections will loss in performances since it will have more operations to treat.

Simulated Traces

. Our framework implements a generator that enables to produce randomized traces by controlling one of the following characteristics :

(1) the proportion of user operation which are insertion, (2) the proportion of user operation which concern blocks, (3) the number of operations generated, (4) the number of replica that will produce operations, (5) the probability at each iteration that a replica produce one operation, (6) the average delay (in number of iteration) between generation of user operation and

[3]We consider branching as concurrent editing.

reception of corresponding operation in other replicas. (7) average size of copy/past.

The simulation ensures that each replica receives operations in the order as defined in the logs. The framework lets replicas of every algorithm generate operations in its own formats for the given trace operations provided from the simulated logs.

3. Synchronous Experiment Results

In the following, we present the experimental results retrieved through our framework, by using synchronous corpus.

Firstly, we present the different corpus obtained and some experiments made as an example. Secondly, we present a performance evaluation of some algorithms for different data types.

The experiments were performed on virtual machines of Amazon Elastic Compute Cloud (EC2). We run each experiment a virtual machine with Intel(R) Xeon(R) 5160 one-core processor (4096K Cache, 2.27GHz, 2266.746 MHz FSB, 3.75 GiB of memory), that has installed GNU/Linux ubuntu 12.04.

3.1. Corpus obtained

Real synchronous Traces

. Based on the characteristics presented in Table 1, we parameter the simulator to produce the same traces. It also provides a controlled simulation environment that replays a trace of operations and measures the performance of the replicated algorithms. The simulation ensures that each replica receives operations in the order as defined in the logs.

Simulated Traces

. To evaluate the algorithm's performance, we produced through the framework several sets of traces. In each experiment, the framework controls one characteristic and keep the others static at values similar to the [1] experiments.[4]. Table 2 resume our experiment.

3.2. Performance in Set Data Type Algorithms

The performance of different algorithms designed for *set* data type are presented in this section. Even if the framework supports CvRDT and CmRDT (Section), we present as an example in this paper only results of CmRDT.

Since the different algorithms have different semantics, we run each algorithm one hundred time and each time in different simulated data (except for OR-set and OptORSet in the same data since they have the same semantics). In the following, we present the results of two experiments: by changing the size of the data set and the percentage of insertions. In Figure 4(a) and 4(b) we present respectively the average execution time for each algorithm during the generation and integration of operations. The results of execution time indicate that:

- OptORSet and ORSet algorithms are the worst in execution time. In local, both algorithms take time to generate a unique tag for each element. While, during the integration, both algorithms treat each remote operation as a new element since they have a different tag,
- OptORSet takes more time than ORSet in local, since it updates the local vector while ORSet algorithm does not,
- LWWSet and CounterSet outperforms ORSet(s) algorithms. Indeed, they don't need to generate a tag for elements,
- LWWSet takes slightly more time than CounterSet. This is due to the management of the timestamp for each element. Indeed, LWWSet traverses its model to find the previous value of the element, and increment it. While, CounterSet algorithm generates directly a counter.

In Figure 5(a) and 5(b) we present respectively the memory requirement for each algorithm per percentage of insertion and size of data set. The performances obtained indicate that:

- Contrary to what we expect, OptORSize does not reduce the memory size. Indeed, the vector used to detect the partial order of elements hide the memory saved,
- The memory used by LWWSet and CounterSet algorithms remains stable. Whatever the percentage of insertion, these algorithms keep all elements (inserted and deleted),
- More the percentage of insertion grows and more the replica lost in memory. Gradually, the memory becomes stable since the elements will be existed in the set,
- The memory requirements are proportional with the size of the data set. Indeed, the number of elements in the replica set depends on the size of data used.

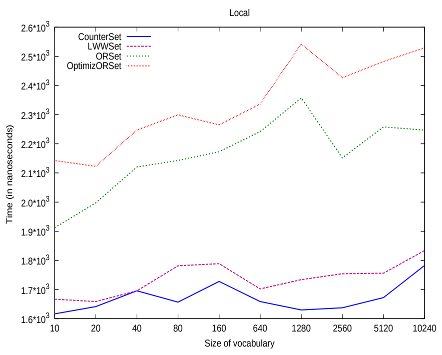
To validate this analysis and make sure that this difference is significant, we use ANOVA technique[5].

[4] To simplify algorithm analysis, we do not modify the delay and generation probability. Also, these characteristic mostly affect the concurrency degree which is also affected by the number of replicas.

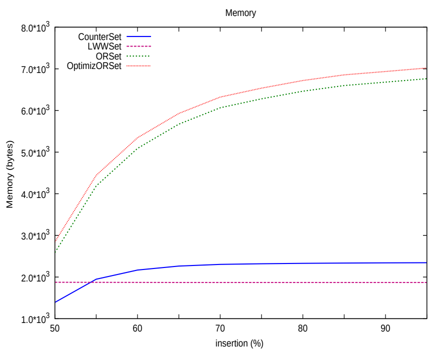
[5]one-way analysis of variance. The result is significant if $p - value$ <.05

Experiment	Operations	Insertions	Blocks for text	vocabulary for set	Replicas
Operations	20 000 $\xrightarrow{+100000}$ 40 000	10 000	10 000	10 000	10 000
Replicas	10	10	10	10	2 $\xrightarrow{\times 5}$ 50
Blocks %	15%	15%	0% $\xrightarrow{+10\%}$ 100%	-	15%
Insertions %	80%	50% $\xrightarrow{+5\%}$ 100%	80%	80	80%
vocab. size	-	-	-	10 $\xrightarrow{\times 2}$ 10240	-
Avg blocks size	100	100	100	-	100
Generation w.p.	0.1	0.1	0.1	0.1	0.1
Delay	5	5	5	5	5

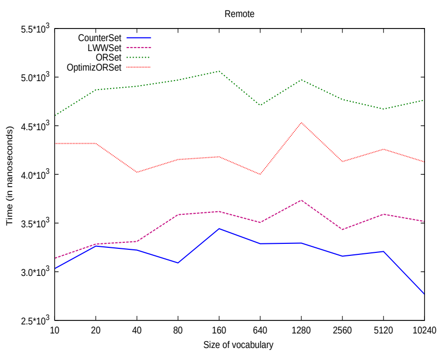
Table 2. Experiment for Text/Set data type algorithms.



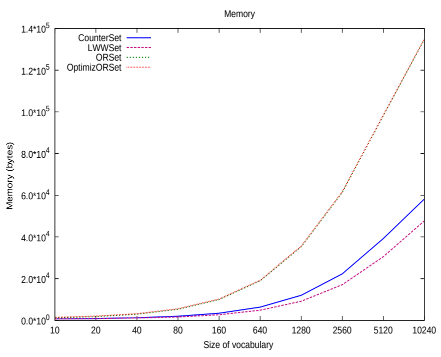
(a) Local execution time



(a) Memory requirement per % insertion



(b) Time of integration



(b) Memory requirement per size of data set

Figure 4. Execution time in set data type

Figure 5. Memory occupation in set data type

The $(p - value)$ obtained from the results of algorithms, demonstrate clearly that all results are very significant. Indeed, for execution time, $p - value = 0.00$ and for memory $p - value = 0.03$. In both experiments, $p - value < 0.05$.

3.3. Performance in Text Data Type Algorithms

In [1], we evaluated through our framework different collaborative editing algorithms in real traces described above. However, in this paper we extend the performance evaluation with two other algorithms that we did not in [1]. In addition, we perform another kind of performance evaluation such as memory requirements and size of messages in different large corpus.

In this section, we present the results of two experiments: i) the performance of algorithms on simulated data inspired from collecting *synchronous* real traces, ii) algorithm's performance in a real *asynchronous* traces obtained from github. As an example, we measured the performances of six algorithms: SOCT2/TTF, WOOTH, RGA, TreeDoc, Logoot, and LogootSplit. We conducted four experiments, each one varying a different characteristic of the simulated traces. To obtain a significant results, we ran each experiment five times.

In this section, we conducted an experiment in two different traces: synchronous and asynchronous. In [44] the authors evaluated Logoot algorithm on Wikipedia traces, and assume that such histories as traces extracted from synchronous collaborative editing. Also, in [26] Treedoc algorithm was evaluated on SVN repositories. In this experiment, we evaluate the algorithms on Git repositories to validate the result of synchronous experiment.

To understand the behaviors of algorithms during the experiment, we conduct in the following a theoretical evaluation before presenting the results.

Theoretical Evaluation

. In [1], we provided a comparison in time complexity of Logoot, WOOTs, RGA and SOCT2 algorithms (Section 2.1). In the following, we specify the time complexity of two other algorithms that we do not presented in [1]. These algorithms are Treedoc and LogootSplit. In contrary to the previous algorithms that are based on character operations, Treedoc and LogootSplit support block granularity.

In addition, in the previous work [1], we do not provide a comparison of space complexity. Then, we presented also a comparison of all algorithms described above in space complexity.

Average-case Time Complexity Analysis Considering the execution time complexity we differentiate the local execution time – treatment of a user operation – and the remote execution time – treatment of a remote operation. The dissemination mechanism is not taken into account since it is independent of the merge mechanism.

During local execution, the algorithms treat either an insert operation (*ins*) or delete operation (*del*). These operations may concern a continuous sequence of elements (e.g. cut or paste a block of text). The algorithms must find in their inner model the position of the user operations. During local operation execution the remote operations that will be sent to other replicas are generated. During remote execution, operations are specific to each algorithm. The average case complexity for each of the above described algorithms is presented in 4. We denote by :

- R the number of replicas,
- H the number number operations that had affected the document,
- N the total number of inserted elements, in worst case $N = H$,
- c the average number of operations concurrent to a given one,[6] in worst case $c = H$,
- n the size of the document view ,
- k the average size of Logoot identifier : in best case $k = 1$ and in worst case $k = N$,
- p the average length of TreeDoc paths : in best case $p = 1$ and in worst case $p = N$,
- l the size of elements present in one operation (blocks), in worst case $l = 1$ or $l = N$ depending if the algorithm manages continuous sequence or not,

Algorithm	Avg. local		Avg. remote	
	ins	del	ins	del
LogootSplit	$\mathcal{O}(H)$	$\mathcal{O}(H)$	$\mathcal{O}(H.log(H))$	$\mathcal{O}(H.log(H))$
TreeDoc	$\mathcal{O}(H)$	$\mathcal{O}(H^2)$	$\mathcal{O}(H)$	$\mathcal{O}(H^2)$

Table 3. Worst time complexities

Algorithm	Avg. local		Avg. remote	
	ins	del	ins	del
LogootSplit	$\mathcal{O}(n/l+k)$	$\mathcal{O}(n/l+k)$	$\mathcal{O}(k.log(n/l))$	$\mathcal{O}(k.log(n/l))$
TreeDoc	$\mathcal{O}(c.p+l)$	$\mathcal{O}(l.p)$	$\mathcal{O}(c.p+l)$	$\mathcal{O}(l.p)$

Table 4. Average time complexities

Both algorithms have its computational complexity impacted by l, the average number of elements present in user operations.

- LogootSplit algorithm uses identifier of $\mathcal{O}(k)$ average size. Its model contains n/l blocks, and the local operations must sum up the block sizes to retrieve operation effect position. The local insertion generates n/l new identifier, while the remote complexity is impacted by the binary search upon identifier in $\mathcal{O}(k.log(n/l))$ average complexity [3].
- TreeDoc algorithm must follow one path of length p in the tree to find user operation effect position and remote operation identifiers. Each node along this path which is a super node [26] must be linearly traversed. So there will be $c.p$ in average, nodes traversed. *ins* operations of a continuous sequence produce only one remote operation while *del* operations produce l remote operations.

[6] c depends on the network latency and user operations frequency.

Space complexity We establish the space occupied by the models used by each algorithm and the size of the message composed of remote operations corresponding to one user operation.

- *SOCT2/TTF* stores the history of operations $\mathcal{O}(H)$ each one with a vector clock $\mathcal{O}(R)$. The l remote operations contains also a vector clock $\mathcal{O}(R)$.
- *WOOT* algorithm family and *RGA* store $\mathcal{O}(N)$ elements with their fixed-size identifiers. A user operation produces l remote operations.
- *Logoot* stores $\mathcal{O}(n)$ elements with their identifier of size $\mathcal{O}(k)$. A user operation produces l remote operations with identifier. A user operation produces l remote operations.
- *LogootSplit* stores only $\mathcal{O}(n/l)$ elements. A user insert operation produces one operation of size $\mathcal{O}(k+l)$: an identifier and a content. A user delete operation produce in average n/l^2 – the number of elements divided by l – remote operations with an identifier.
- *Treedoc* stores a tree of $\mathcal{O}(n)$ elements. A user insert operation produces one operation of size $\mathcal{O}(p+l)$. A user delete operation produces l remote operations with identifier in $\mathcal{O}(p)$.

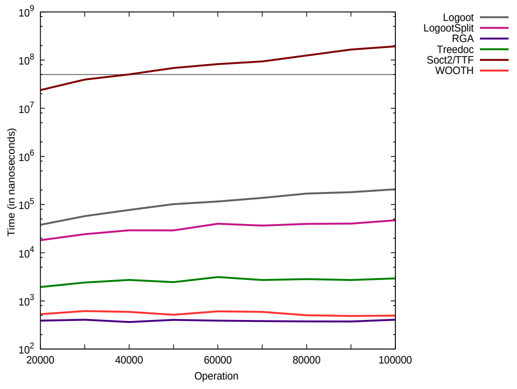
Algorithm	model	message
SOCT2/TTF	$\mathcal{O}(H.R)$	$\mathcal{O}(l.R)$
WOOTs	$\mathcal{O}(N)$	$\mathcal{O}(l)$
RGA	$\mathcal{O}(N)$	$\mathcal{O}(l)$
Logoot	$\mathcal{O}(k.n)$	$\mathcal{O}(l.k)$
LogootSplit	$\mathcal{O}(k.n/l)$	$\mathcal{O}(k+l)\|\mathcal{O}(k.n/l^2)$
TreeDoc	$\mathcal{O}(n)$	$\mathcal{O}(p+l)\|\mathcal{O}(p.l)$

Table 5. Space complexity analysis of meta-data

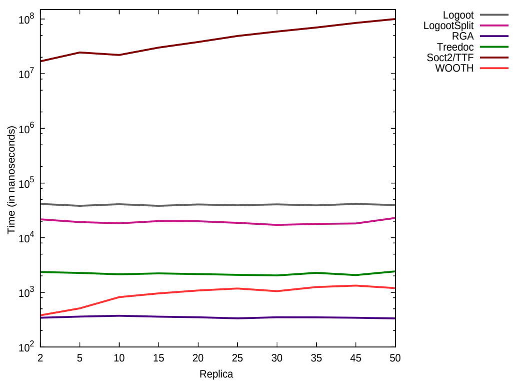
Discussion The theoretical analysis shows that no algorithm performance surpass all the others whether in worst case or average case. However, RGA and TreeDoc seem good candidates, but must be compared with each other and with LogootSplit which follows a different approach. Since the theoretical analysis is not sufficient to rank the different algorithms, an experimentation is required. Why, an implementation of a framework that allows an experimental evaluation is needed. In the following section, we present a framework to experiment the algorithms in realistic settings.

Remote execution time

. The figures 6(a) and 6(b) present the performance of the remote operations execution in two different experiments: number of operations and number of replicas. The figures vertical axis presents the average time taken to execute the remote operation – or the set of remote operations – corresponding to a user operation appearing in the trace. This axis uses a logarithmic scale. The horizontal axis presents the experimental parameter value, one label per trace.



(a) Number of operations



(b) Number of replicas

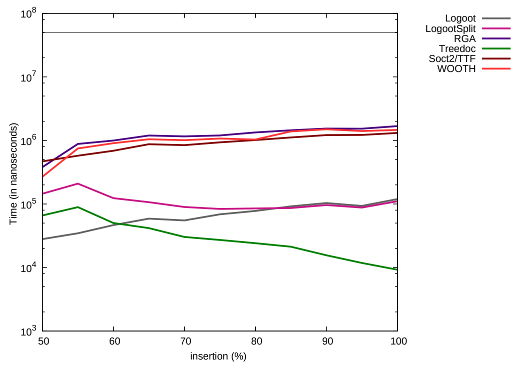
Figure 6. Remote execution time in text data type

Operations experiment WOOTH and RGA perform very stably. Logoot performance decrease when the number of operations grows. This is due to a growing identifiers for Logoot. Identifier growing also affect TreeDoc and LogootSplit but only slightly.

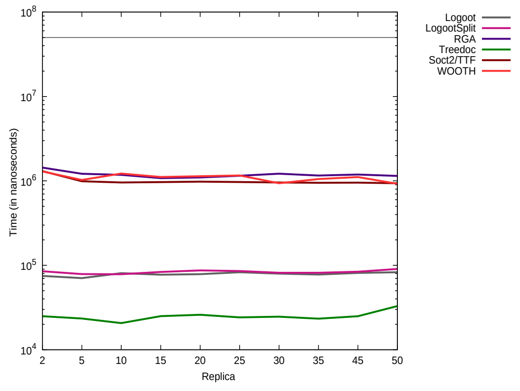
Replicas experiment We can observe that the CRDT algorithms perform very stably when the number of replica increases. The SOCT2/TTF performance eventually degrades but less quickly than the number of replicas. This loss is due to increased c concurrency and not the number of replicas, since the algorithm does not traverse the vector clocks during remote execution.

Local execution time

. The figures 7(a) and 7(b) present respectively the average local execution time for the user operations by insertions proportion and replica numbers. The vertical axis uses a logarithmic scale. The execution of a user operation includes the creation of the corresponding remote operations.



(a) Local execution



(b) Number of replicas

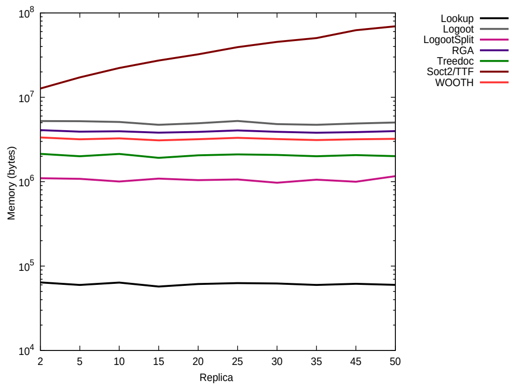
Figure 7. Local execute time in text data type

Insertion experiment The algorithms using tombstones (RGA, WOOTH and TTF) perform worst when the proportion of insertion grows. Indeed, the must traverse their model – which becomes slightly larger – to find user effect operation position. Logoot performs worst due to growing Logoot identifiers. The TreeDoc and LogootSplit algorithms performances have improved since, in both algorithms, an insert operation requires only one treatment per block contrary to delete operations.

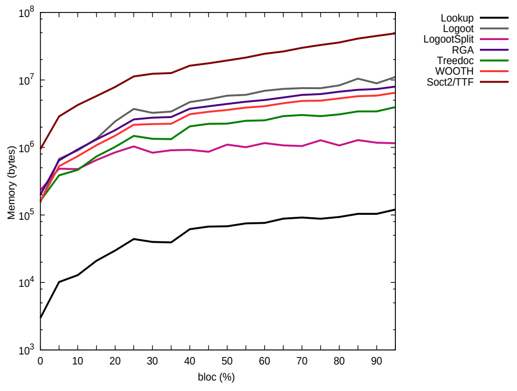
Replicas experiment As for remote execution, TreeDoc, Logoot and LogootSplit perform very stably facing the increasing number of replicas. RGA and WOOTH sightly loss in performance due to increased concurrency. In contrary to remote execution, SOCT2 remains stable since it does not use the vector in local.

Memory requirement

. In this subsection we present algorithm's performance concerning memory usage. Figure 8(a) presents the average memory, computed using Java serialization, required by one replica during the replicas experiment. The results also include the average size of the document (Lookup), i.e. the size of the view presented to the user. The difference between the document size and the replica size indicates the overhead of an algorithm.



(a) Number of replicas



(b) proportion of blocks

Figure 8. Memory occupation in text data type

As for execution time, the number of replicas does not affect the memory required by the algorithms, except for SOCT2/TTF since it uses vector clocks to detect concurrent operations. All other algorithm behaviors remains very stable during the whole experiment. The worst algorithm is SOCT2/TTF that requires tombstones and buffer history. More surprisingly, and contrary to [44] experiments that use a line granularity, the second worst algorithm is Logoot which does not require tombstones but non-bounded identifier for each character. Logoot requires an overhead of one hundred times the document's size. The following are, in order, RGA, WOOTH, TreeDoc, and LogootSplit. Despite having the best performance, LogootSplit requires an overhead of a twenty times the document's size.

Figure 8(b) presents the average memory occupied by one replica during the block experiment. Except LogootSplit, the memory occupied increases for all algorithms and for the document view, since more blocks implies more characters. The ranking between the algorithms and their relative overhead remains the same than in the replica experiment. LogootSplit performance has a different behavior. With 0% block it has similar performance than the other. With more than zero block, its performance is stable even if the document size increases. With 60% block, its overhead is only three times the document size, far below the other algorithms.

Message size

. To evaluate the bandwidth required the algorithms, we compute the size of the remote messages generated by the algorithms. Figure 9 presents the average size of the message serialized depending on proportion of operations that manipulate the blocks. Not surprisingly, the behavior and the ranking between the algorithm is the same than for the model size results. However, in the experiment with 0% block, TreeDoc performs worse compare to all other CRDT algorithms and as badly as SOCT2/TTF. Indeed, in the TreeDoc model the identifiers representation is compacted into the tree, but not into the remote message. Over the time, SOCT2/TTF produces an important flow in the network. The algorithm sends in its messages the vector clock of the current operation. That is why, SOCT2/TTF does not scale.

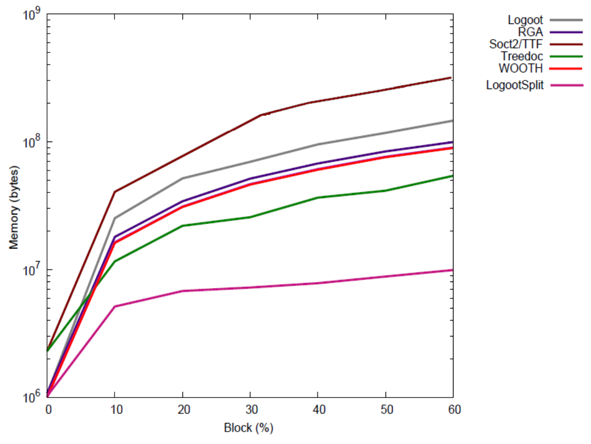


Figure 9. Size of messages in text data type

3.4. Performance in Tree Data Type Algorithms

In [19], we measured through the framework a performance evaluation of tree data type algorithms: TreeOpt, OTTree and FCEdit algorithms. We evaluated these algorithms with different policies described in section 2.1, and compared them with other algorithms deployed on layered approach[19].

3.5. Discussion

The results obtained trough our framework are consistent with the theoretical analysis. Such consistency validates the evaluated implementation of the algorithms. Despite the random nature of our different experiments, their results are consistent with each other.

Our experiments demonstrate that our framework is able to detect which parameter has more effect on the algorithms. In addition, Analyzing the results obtained by our framework, we were able to select which algorithm is adapted to which situation.

The framework provided also for the first, an experimental evaluation of set data type algorithms, and provided a more performance evaluation of text data type algorithms than published in [1]. For instance, study the impact of execution time by number of replicas, the memory requirement and sizes of messages for text data type has never been studied.

As shown by our experimental results, the choice of the best algorithm depends highly on the application and user context. For the set data type algorithms, CounterSet is the best choice in execution time and for application with a large dataset. While, LWWSet algorithm is good for applications that require low memory. We found also, OptORSet algorithm designed to reduce the memory requirement does not. Indeed, the vector used to detect the partial order of elements hide the memory saved.

For text data type algorithms, TreeDoc is a very good generic algorithm. It performs the best or among the best for almost all experiment. However, we cannot recommend it for every usage. For collaborative activities with a very large number of participants, RGA and WOOTH are more suitable. Indeed, their remote execution times are the best. For usages on limited devices, LogootSplit is a good choice since it has a low memory and bandwidth overhead and good execution time performance.

4. Asynchronous Editing Experiment

In asynchronous editing, the documents are modified in a different manner compared to synchronous editing. Operations are grouped in patches and the users usually control when they apply remotes modifications, introducing more concurrency. In this Section, we evaluate the behavior of the studied collaborative editing algorithms in the context of asynchronous collaboration.

As explained Section 2.2, our framework is able to extract collaborative editing corpus from git repositories and to run a different algorithm against these histories. These experiments are run in the same Amazon EC2 setting than above.

We use different statistical metrics to evaluate the algorithms. We establish the average performance of an algorithm to manage three whole repositories and to treat a user operation: commit or merge. We analysis the correlation between their performances and the edition characteristics per file. Finally, we evaluate the quality of the automatic merge result.

4.1. Corpus obtained

In Table 6 we present the characteristics of some projects retrieved from the GitHub repository. The head commits sha1 used to run our experiments is presented above each repository name. The characteristics are computed per file. For each file, we extracted the number of commits that affected the file, the total number of lines modified, the number of simulated replicas, the proportion of insertions, the average block size in number of lines (as computed by the diff), and the number of merge commits. These repositories are selected among the most popular projects available on GitHub web site.[7]

General performance The Table 7 presents the execution time of the different algorithms on three git repositories (git/git, joyent/node and twitter/bootstrap). The "commit" measure is the time required to execute all the local operations affecting the files in a repository divided by the number of commits. The "merge" measure is the time required to execute all the remote operations affecting the files in a repository divided by the number of commits. Thus, these measures correspond to the relative performance a user should observe in a DVCS based on one of these algorithms.

In local execution, the ranking between the performances of the algorithms is the roughly the same than in the synchronous experiment. we can classify the algorithms on three sets: algorithms based on blocks which perform better – LogootSplit and Treedoc – algorithms based on tombstones which perform worse – WOOTH, RGA and SOCT2/TTF – and Logoot somewhere in the middle. However, the rankings within these sets can vary due to collaboration characteristics. For instance, LogootSplit seems more stable than TreeDoc, and RGA seems more stable than WOOTH.

[7]https://github.com/popular/starred

In remote execution, the results are more surprising compared to the synchronous experiments. SOCT2/TTF performances remain the worst, requiring hundreds time more than the average CRDT. RGA, TreeDoc, Logoot and LogootSplit performance remains very good. However, contrary to asynchronous experiment, Logoot performs slightly better than LogootSplit. Also, WOOTH behavior was not expected. It is the worst CRDT algorithm on bootstrap and node project. This behavior seems due to few of the most edited files in these projects that cause problems to the WOOT linearisation algorithm. Indeed, Table 6 shows that these two projects contains files which are much more edited than the git one.

Tendencies To detect which characteristic affect the algorithms performances, we correlate the performance to the editions characteristic. We consider the independent performances result of each file edited of one repository: git/git.[8] The presented results consider only the most edited files, i.e. more than the average which is 60 block edited.

We computed the bivariate analysis correlation between each local and remote execution time and four characteristics. The execution time is the average execution time per line edited. The four characteristics are the average size of blocks, the proportion of insertion operations, the number of lines, and the number of simulated replicas. We noticed that the execution times correlate strongly with the average size of blocks. So, we computed the partial correlation between execution time and each other characteristic controlling the block size.

4.2. *Remote Editing Experiment*

Table 8 presents the correlations between remote execution time and edition characteristics. Figures 10(a) and 10(b) present respectively, the scatter graph for execution time and number of lines and number of replicas. The execution time is presented on logarithmic scale.

Algorithm	Lines		Replicas	
	Bivariate	Partial	Bivariate	Partial
SOCT2/TTF	0.99	0.99	0.45	0.3
WOOTH	0.07	0	0.07	0.03
RGA	-0.06	-0.12	0.02	0.03
Logoot	0.18	-0.01	0.07	-0.06
LogootSplit	0.34	-0.25	0.075	-0.02
TreeDoc	-0.01	-0.08	0.02	0.02

Table 8. Correlation on remote execution time.

[8]The two other projects contains less files and/or few heavily edited files which makes the tendencies more difficult for analysis.

Project	git/git			twitter/bootstrap			joyent/node		
Head sha1	8c7a786b6c8			cd42f56178			9f29785783		
Files	2496			250			2588		
Characteristics	max	min	avg	max	min	avg	max	min	avg
Commits	1742	2	75.75	633	12	84.21	631	2	21.75
Merge	451	0	12.92	55	0	7.44	37	0	0.88
Lines edited	25304	149	1953.49	119594	964	18104.65	123340	276	7400.23
Avg block size	28.13	1.84	6.78	80.93	12.35	32.06	761.25	7.65	97.45

Table 6. Projects characteristics

Algorithm	git		node		bootstrap	
	Commit	Merge	Commit	Merge	Commit	Merge
SOCT2/TTF	337	6723	1816	18153	1373.0	33485.8
WOOTH	306	15	2232	133	1936.9	612.4
RGA	365	13	1662.2	49.3	1709.1	64.1
Logoot	159	24	1138.2	64.7	1337.0	76.0
LogootSplit	99	63	135.2	90.1	188.5	97.7
TreeDoc	96	18	200.4	41.9	595.0	76.1

Table 7. Average commit and merge time (in μs)

The table and the figures obtained through the framework demonstrates that

- the tendency remote performances and ranking between the algorithms are consistent with synchronous editing experiments;
- all CRDT performances are very stable against all edition characteristics with negative or very low (under 0.1) partial correlations;
- the SOCT2 algorithm remote performances are impacted by the number of operation and number of replica;
- LogootSplit gains in performance during the edition process – number of lines, -0.25 partial correlation.

4.3. *Local experiment*

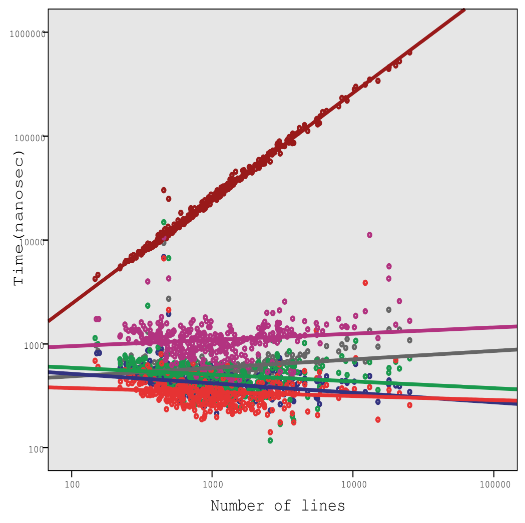
The visual analysis of the scatter charts is less obvious than for remote execution. Why, statistical analysis of the data may require to ensure consistency with our other experimental results. The analysis of the local execution time is presented in Table 9.

Algorithm	Blocks	insertion		Replicas	
		Bivariate	Partial	Bivariate	Partial
SOCT2/TTF	0.69	0.16	0.32	0.11	-0.12
WOOTH	0.83	0.02	0.29	0.34	-0.01
RGA	0.77	0.09	0.32	0.21	-0.11
Logoot	0.51	0.24	0.33	-0.01	-0.01
LogootSplit	0.87	-0.09	0.01	0.15	-0.09
TreeDoc	0.68	-0.15	-0.15	0.13	0.13

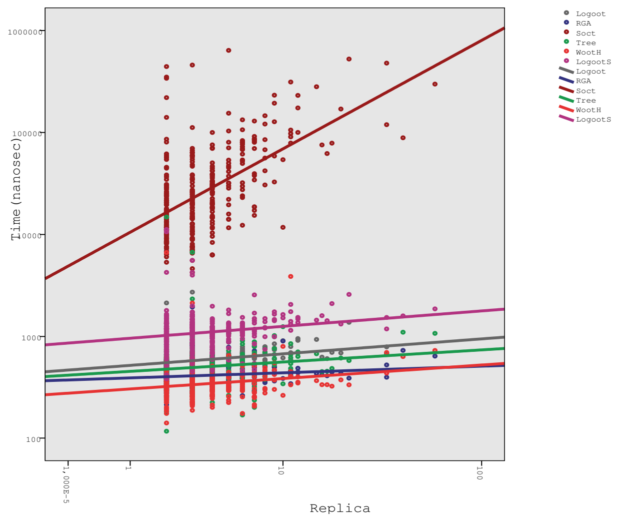
Table 9. Correlation on local execution time

The correlation values indicate that:

- The local performances and ranking between the algorithms are consistent with synchronous editing experiments;
- The performance of all algorithms does not affected by the number of replicas;
- Treedoc algorithm has the best performance. It improves the performance since it identifies each line by unique identifier and the size of major-nodes decreases by the percentage of insertion. However, LogootSplit remains stable since all operation on Git are based on line. However, Logoot algorithm and the algorithms based on tombstones (RGA, WOOTH and SOCT2) are less efficient;
- The partial correlation shows that the insertion proportion impact tombstone-based algorithms (similar to synchronous experiments);



(a) Remote execution -number of lines-



(b) Remote execution -number of replica-

Figure 10. Remote execution time

- All algorithms, including SOCT2, performs very stably when the number of replicas grows. The partial correlation in Table 9 is around 0 for all algorithms;
- As for remote execution, the hierarchy between the performance scalability of LogootSplit and TreeDoc is less obvious than in synchronous experiment.

4.4. *Memory requirement*

Table 10 presents the total memory occupied by the algorithms during the experiments and their average message size. We also present the space occupied by the GIT software in its .git directory. The obtained results are consistent with the synchronous experiment. The ranking between the algorithm is respect except than Logoot performs relatively better in the asynchronous one. The algorithms based on tombstones – SOCT2/TTF, RGA and WOOTH in this order – have the worst performances since they do not delete the elements. TreeDoc and LogootSplit obtain the best performances.

CRDT algorithms memory is lower or comparable to the corresponding .git directory size despite the poor performances of the Java serialization mechanism and the compression made by the git software.[9] However, git software is able to retrieve any past states of the document. This feature is achievable in OT and in CRDT by using tombstones, thus, without supplementary memory cost for RGA and WOOTH. Indeed, any element identifier in a CRDT model contains a clock.

4.5. *Message size*

The message size of the algorithms in asynchronous traces presented in table 10, follow the same ranking to the one obtained through synchronous experiment. Indeed, the worst algorithm is SOCT2. It produces the largest message size since it sends the vector clock. SOCT2 does not scale. RGA and WOOTH specifies the previous and next element in their messages, while Logoot adds its identifier. Treedoc and LogootSplit that are based in blocks are the best, since their identifiers are not voluminous.

4.6. *Merge quality*

In [2], we proposed a methodology to evaluate the algorithms in merge quality. The framework reproduces the same collaboration as in the histories of DVCS and observe the effort made by users when coniňĆicts occur. The framework returns the number of blocks that conflict and number of lines introduced by users to correct their document. Depending of the algorithms, the collaboration produces less conflict and then user makes less effort to correct the document.

4.7. Discussion

This experiment on asynchronous traces validate the result obtained in synchronous experiment. The hierarchy between the algorithms in term of local/remote performances and messages size is similar to the one obtained through synchronous experiment. The framework re-plays the traces in the same conditions and uses the same technique to compute the performances.

[9]git software stores supplementary information concerning management of directory states. However, this information can be considered much smaller than file states.

Algorithm	git/git	twitter/bootstrap	joyent/node	messages
.git	60	26	171	-
SOCT2/TTF	125.36	39.50	223.51	147.40
WootH	55.84	18.38	88.14	107.85
RGA	63.94	21.41	101.24	116.87
Logoot	45.21	8.10	67.30	113.58
LogootSplit	29.22	6.32	44.98	66.105
Treedoc	38.69	7.61	22.85	73.40

Table 10. Total memory occupation (in mo)

The tool computes also the developer effort in terms of number of modifications that have to be made to correct their document in the merged document. Our framework allows also to evaluate, compare and improve merge algorithms. This is an important information in the software development field where the concurrent edition of document represents a large and fundamental part of the activity.

Using our tool and method, one can evaluate and compare more complex algorithms such as syntactical or semantical merge tools.

5. Related Work

With the development of Web 2.0, the explosion of Internet-based and peer-to-peer services, achieving eventual consistency for large scale distributed systems becomes difficult to achieve. Studied the techniques to achieve eventual consistency and offer a tool to evaluate them have been an active focus point in recent research. However, until now there is no public tool that allows researchers to compare and evaluate their techniques.

A first evaluation of eventual consistency algorithms was performed for *text* data types and was presented in [17]. However, this work was made on unknown tool. In addition, they consider only one variation of OT algorithm and not CRDTs approaches.

In [30] OT algorithm (called ABST) suitable for intermittent connections in mobile devices was proposed and its evaluation was provided. However, the algorithm was compared with only one another algorithm (ABT), the researchers limited their evaluation with only the ratio of insertions, and the evaluation was made on mobile devices by using a closed source implementation. Treedoc [26] and Logoot [44, 45] were evaluated on memory usage using centralized asynchronous traces but not on execution time. RGA is evaluated on [28] but not compared with other algorithms. WOOT and RGA algorithms were never evaluated on their memory requirements. However, all these experiments were made independently and there is no framework that allows a comparison in the same environment.

In [1], we deployed our framework and we evaluated for the first time a comparison of different text data type algorithms in execution time, and by using a real synchronous traces. In this experiment we did not assess the memory space required and messages size produced.

In [19], we evaluated also through our framework the performance of the different *tree* data type. However, we presented just the results without explaining in detail what the framework provides.

At this time, the framework was not able to use an asynchronous traces.

In [2], we evaluated the text data type in asynchronous traces. We used the framework to observe the cases where the conflict occurs and after we proposed a methodology to evaluate the algorithms in merge quality.

For the *set* data types, in [7] the authors improved OR-Set algorithm and presented the result of its performance. The authors implemented a Java client library and perform the experiment in the real database, but the tool used remains private. Until now, the different set data type studied have not been compared with each other, and there is no tool which able to compare them.

In this regard, this is the first paper that presents an open-source framework that allows an evaluation of different eventual consistency algorithms. The framework provides a simulator to produce traces and a mechanism to extract and re-play the histories of Git. In addition, the framework allows users to control different parameter and observe the behavior of algorithms. All algorithms integrated were written in the same language. The framework computes the performance of algorithms such as execution time, memory occupation, memory requirement, messages size and quality of merges.

6. Conclusion and Future work

Achieving consistency in large-scale distributed systems is not an easy task. Since the CAP theorem [6, 10] which states that any distributed computer system cannot provide simultaneous guarantees for consistency (C), availability (A) and to tolerate network partition (P), many decentralized algorithms for different data types are developed and claim that ensure eventual consistency. However, evaluation of such algorithms, compare with each other and know which algorithm is suitable for which situation, require tools that provide a same experimental environment and guarantee consistent results.

In this paper, we *proposed a framework to automatically measure the performances of different decentralized EC data type algorithms*. The framework provides a simulator to generate traces in synchronous and asynchronous mode, and allow to each algorithm to specify the operation on their own format. The framework provides also a mechanism to extract the DVCS history to simulate an editing session.

Moreover, the framework gives to the user the entire control of different parameters during the simulation. Depending of these factors, the framework allows to detect which factor most affects the algorithms performances. It helps to select which algorithm is the most suitable for which situation.

We found out that CRDT algorithms are suitable for synchronous and asynchronous application. Moreover, they outperform some representative operational transformation approaches that were well established for real-time collaboration. However, We found that the choice of algorithms depends on the target of the application and the type of collaboration.

For the set data type algorithms, CounterSet is the best choice in execution time and for application with a large dataset. While, LWWSet algorithm is good for applications that require low memory. We found also, OptORSet algorithm designed to reduce the memory requirement does not. Indeed, the vector used to detect the partial order of elements hide the memory saved.

However, for text data type algorithms, we found that the major factor that affects the algorithms based on tombstones are the size of insertions and number of operations in addition to the number of replicas for OT algorithms. We found also that the algorithms based on blocks are more suitable for state-based approaches than operation-based approaches. indeed, in state-based approaches, the operations are composed of commit and merge operations (treated by lines), Whereas operation-based approaches are based on characters granularity. On memory, LogootSplit and Treedoc are the most appropriate for the both approaches. OT algorithms take the worst performance, it exceeds 50 ms in real-time collaboration. Following [32], SOCT2/TTF is not suitable for real-time collaboration. In addition, it occupies much memory to store all elements. SOCT2/TTF algorithm cannot be applied on application as mobile since it requires a large performance.

In this paper, we conducted different experiments for different eventual consistency algorithms. Firstly, we gave an overview of the framework. We studied how the framework generates traces, presented the different data type supported and the different algorithms implemented. Secondly we conducted two experiments in synchronous and asynchronous mode and we analyzed the performance results. The results obtained are consistent and significant in synchronous and asynchronous traces, and compatible with the theoretical analysis. Such consistency validates the evaluated implementation of our framework.

The framework is built upon an open-source performance evaluation tool and will be itself open-sourced[10]. Thus, eventual consistency tool developers will be able to evaluate and select the suitable algorithm for their needs. Also, computer scientists who design EC algorithms will be able to evaluate and to compare with others.

We plan to extend our framework by adding a mechanism to manage file system traces.

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 : L. André (LogootSplit), G. Oster (SOCT2), H.-G. Roh (RGA), M. Zawirski (TreeDoc) and S. Martin (measurements).

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, P. Urso, and F. Charoy. Improving textual merge result. In *CollaborateCom*, pages 390–399, 2013.

[3] L. André and G. Oster. LogootSplit. Publication in preparation, february 2012.

[4] R. Baldoni and M. Raynal. Fundamentals of distributed computing: A practical tour of vector clock systems. *IEEE Distributed Systems Online*, 3(2):12, 2002.

[5] A. Bieniusa, M. Zawirski, N. M. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. Brief announcement: Semantics of eventually consistent replicated sets. In *DISC*, pages 441–442, 2012.

[6] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[7] A. Deftu and J. Griebsch. A scalable conflict-free replicated set data type. In *Distributed Computing Systems*

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

(ICDCS), 2013 IEEE 33rd International Conference on, pages 186–195, July 2013.

[8] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *SIGMOD Conference*, pages 399–407. ACM Press, 1989.

[9] T. A. S. Foundation. Apache CouchBD. http://couchdb.apache.org/.

[10] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[11] C.-L. Ignat and M. C. Norrie. Customizable collaborative editor relying on treeopt algorithm. In *Proceedings of the eighth conference on European Conference on Computer Supported Cooperative Work*, ECSCW'03, pages 315–334, Norwell, MA, USA, 2003. Kluwer Academic Publishers.

[12] 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.

[13] G. Inc. Google inc. what's different about the new google docs. Webpage, 2010.

[14] P. R. Johnson and R. H. Thomas. RFC 677: Maintenance of duplicate databases, January 1975, (Septembre 2005). http://www.ietf.org/rfc/rfc677.txt.

[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] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[17] D. Li and R. Li. A performance study of group editing algorithms. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 1, page 8 pp., 0-0 2006.

[18] S. Martin, M. Ahmed-Nacer, and P. Urso. Abstract unordered and ordered trees crdt. Rapport de recherche RR-7825, INRIA, December 2011.

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

[20] S. Martin and D. Lugiez. Collaborative peer to peer edition: Avoiding conflicts is better than solving conflicts. In H. Weghorn and P. T. Isaías, editors, *IADIS AC (2)*, pages 124–128. IADIS Press, 2009.

[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] E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[23] D. A. Nichols, P. Curtis, M. Dixon, and J. 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, pages 111–120, New York, NY, USA, 1995. ACM.

[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, Alberta, Canada, nov 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] 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.

[27] 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.

[28] 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.

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

[30] B. Shao, D. Li, and N. Gu. A Fast Operational Transformation Algorithm for Mobile and Asynchronous Collaboration. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1707–1720, December 2010.

[31] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976, pages 386–400, Grenoble, France, October 2011.

[32] B. Shneiderman. Response Time and Display Rate in Human Performance with Computers. *ACM Computing Surveys*, 16(3):265–285, September 1984.

[33] 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.

[34] M. Suleiman, M. Cart, and J. Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *Proceedings of the fourteenth International Conference on Data Engineering - ICDE'98*, pages 36–45, Orlando, Floride, États-Unis, February 1998. IEEE Computer Society.

[35] 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.

[36] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and

intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108, March 1998.

[37] D. Sun and C. Sun. Operation Context and Context-based Operational Transformation. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, pages 279–288, Banff, Alberta, Canada, November 2006. ACM Press.

[38] TeamEdit. a collaborative text editor. http://teamedit.sourceforge.net/.

[39] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles - SOSP'95*, pages 172–182. ACM Press, 1995.

[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. Wooki: a P2P Wiki-based Collaborative Writing Tool. In *Web Information Systems Engineering*, pages 503–512, Nancy, France, December 2007. Springer.

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

[44] 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.

[45] 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.

[46] S. Xia, D. Sun, C. Sun, D. Chen, and H. Shen. Leveraging single-user applications for multi-user collaboration: the coword approach. In J. D. Herbsleb and G. M. Olson, editors, *CSCW*, pages 162–171. ACM, 2004.