A Comparison of Optimistic Approaches to Collaborative Editing of Wiki Pages

C.-L. Ignat, G. Oster and P. Molli
LORIA, INRIA (Nancy-Grand Est)
{claudia.ignat,oster,molli}@loria.fr

M. Cart and J. Ferrié
LIRMM, University of Montpellier 2
{cart,ferrie}@lirmm.fr

A.-M. Kermarrec
IRISA, INRIA (Rennes-Bretagne Atlantique)
anne-marie.kermarrec@irisa.fr

P. Sutra, M. Shapiro, L. Benmouffok and J.-M. Busca
LIP6, INRIA (Paris-Rocquencourt)
pierre.sutra@lip6.fr

R. Guerraoui
EPFL
rachid.guerraoui@epfl.ch

***Abstract*—Wikis, a popular tool for sharing knowledge, are basically collaborative editing systems. However, existing wiki systems offer limited support for co-operative authoring, and they do not scale well, because they are based on a centralised architecture. This paper compares the well-known centralised MediaWiki system with several peer-to-peer approaches to editing of wiki pages: an operational transformation approach (MOT2), a commutativity-oriented approach (WOOTO) and a conflict resolution approach (ACF). We evaluate and compare them, according to a number of qualitative and quantitative metrics.**

I. Introduction

In recent years, the Web has seen an explosive growth of massive collaboration tools, such as wiki and weblog systems. By the billions, users may share knowledge and collectively advance innovation, in various fields of science and art.

Existing tools, such as the MediaWiki system for wikis, are popular in part because they do not require any specific skills. However, they are based on a centralised architecture and hence do not scale well. Moreover, they provide limited functionality for collaborative authoring of shared documents.

At the same time, peer-to-peer (P2P) techniques have grown equally explosively. They enable massive sharing of audio, video, data or any other digital content, without the need for a central server, and its attendant administration and hardware costs. P2P systems provide availability and scalability by replicating data, and by balancing workload among peers. However, current P2P networks are designed to distribute only immutable documents.

A natural research direction is to use P2P techniques to distribute collaborative documents. This raises the issue of supporting collaborative edits, and of maintaining consistency, over a massive population of users, shared documents, and sites. The purpose of this article is to study a number of alternative P2P, decentralised approaches, applied to collaborative wiki editing, contrasted with current centralised systems. Specifically, we detail P2P broadcast techniques, and we compare the existing centralised approach (MediaWiki) with several distributed, peer-to-peer approaches, namely: an operational transformation approach (MOT2 [2]), a commutativity-oriented approach (WOOTO [9], [18]) and a serialisation and conflict resolution approach (ACF [13]). We evaluate each approach according to a number of specified qualitative and quantitative metrics.

The paper is organised as follows. In Section II we present both the basic concepts of optimistic collaborative editing and our evaluation metrics. Section III presents our running example, concurrent editing scenario of a wiki page. Next, Section IV develops this scenario using MediaWiki. Section V overviews P2P broadcast mechanisms. The next sections discuss and evaluate specific P2P concurrency and consistency techniques, under the same scenario: the MOT2 operational transformation in Section VI; the WOOTO commutativity-oriented approach in Section VII; and the ACF reconciliation approach in Section VIII. A summary is presented in Section IX.

II. Optimistic Replication in Peer-to-Peer Systems

This section presents the main concepts in optimistic replication [12] and some evaluation criteria for the comparison of various optimistic replication approaches.

A. Basic Concepts

A peer-to-peer collaborative editing system is composed of a set of *peers* (sites) that can dynamically join and leave the system. Peers host *replicas* of the shared document. Users generate *operations* to modify the shared data. An operation undergoes the following sequence of events:

1) The user submits it at some site.
2) It is executed against the local replica.
3) It is broadcast through the P2P network to the other peers.
4) In some systems, it is subjected to conflict detection and *reconciliation* with respect to concurrent operations.
5) Peer sites receive it and *integrate* it, i.e., they execute it against their own replica.

Operations are kept in a buffer, called a *log* or *history*. Peers synchronise with one another by exchanging and merging logs, through *epidemic propagation* (see Section V).

An operation is valid under some *precondition*. Preconditions can be implicitly built into the replication algorithm, or can be written explicitly by users.

Additionally, an operation might contain a *postcondition* that should be satisfied after its execution. For instance, consider a document represented as a linear sequence of characters, where each character is uniquely identified. Operation $insert(c, c_p, c_n)$ inserts character c between characters c_p and c_n. Its precondition is that c_p and c_n exist and that c_p is ordered before c_n. A postcondition is that c is inserted after c_p and before c_n.

A common precondition is to maintain some relation with another operation. For instance, in many systems, the order of execution of operations is compatible with the "happens-before" relation [7]. We say that op_1 *happens-before* op_2 if op_2 was generated on some site, after op_1 executed on the same site.

Conversely, op_1 and op_2 are said *concurrent* if neither op_1 *happens-before* op_2 nor op_2 *happens-before* op_1. Concurrent updates of different copies of the same document might generate conflicting changes. A conflict occurs when an operation would fail to satisfy its precondition. Detected conflicts can be resolved automatically by the system or manually by users.

The system should ensure *eventual consistency*, i.e., if all clients stop submitting operations, all sites eventually converge to the same (correct) state.

B. Evaluation Criteria

We will compare different approaches according to a large spectrum of metrics, some qualitative, some quantitative. Parameters are: the number of sites m, the number of operations n, and the size of the edited document l.

a) *Communication Complexity:* The number of messages exchanged for all sites to converge to the final state.
b) *Time Complexity:* This evaluates the time to convergence.
c) *Space Complexity:* The amount of memory required (per site).
d) *First Site Convergence Latency:* The minimal number of rounds necessary for the first site to reach its final state. We define a round as a P2P round-trip of communication. Generally, each peer sends out messages to all other peers, receives all messages sent to it at that round, and carries out some local computation [8].
e) *Convergence Latency:* The number of rounds necessary for *all* sites to converge to the final state.
f) *Semantic Expressiveness:* We say a system is semantically expressive if it can capture a large spectrum of operation semantics, and of preconditions and postconditions.
g) *Determinism:* An approach is said deterministic if the final document state is determined only by the set of operations and their preconditions, and does not depend on order of message delivery.

III. Concurrent Editing Scenario of a Wiki Page

In this section we present an example of users concurrently editing a wiki page. We will use this example throughout this paper to illustrate various optimistic replication approaches for maintaining consistency.

Consider three users collaboratively writing a wiki page about optimistic replication. Suppose the three users concurrently edit the section *"Detection and resolution of conflicts"*, of the wiki page whose initial state is illustrated in Figure 1.

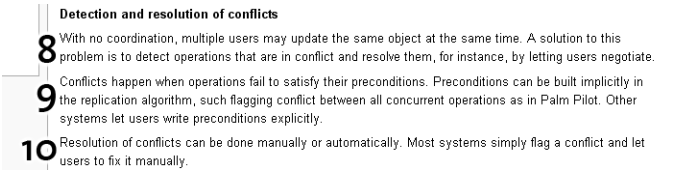

Fig. 1. Initial state of the section *"Detection and resolution of conflicts"*

Further suppose the three users perform the operations : User 1 inserts a new line as the 10th line, User 2 updates the 9th line, User 3 deletes the lines 8 to 10. The modifications of each user are shown in the same figure, Figure 2.

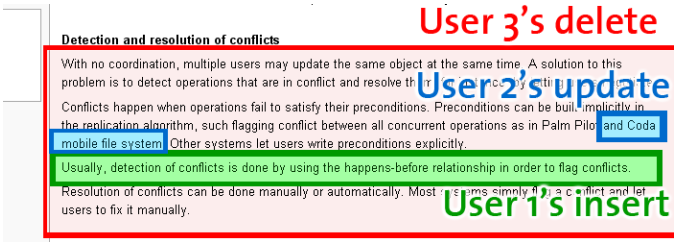

Fig. 2. Overview of changes performed by users.

Afterwards, all three users try to publish their versions of the document by exchanging their modifications. Eventually all replicas should converge.

To ensure this property, a simple technique is the "Thomas Write Rule" approach [16], also called "Last-Writer Wins" in replicated file systems. The successive versions of a file are timestamped or numbered; the version with the highest number is retained, and other versions are thrown away. The drawback of course is that concurrent updates are lost.

Instead, the literature on computer-supported co-operative work says that "user intention" should be preserved. Unfortunately, it is difficult to characterise user intention formally. The systems we review here differ, in particular, in how they capture this concept. Operational Transformation specifies transformations between pairs of concurrent operations, in order to combine their effects. The commutativity-oriented approach defines operations that commute, in order to reach the same result even if operations execute in different orders. The action-constraint framework maintains an explicit representation of semantic relations between operations, e.g., conflicts, in order to combine them optimally.

Step	Actions of User 1	Actions of User 2	Actions of User 3
1	inserts a new line as the 10th line	updates the 9th line	deletes the lines 8–10
2	SAVE	SAVE	SAVE
3	→ save is aborted → conflict is detected (changes of User 1 and User 3 conflict) → both versions are presented	→ save is aborted → conflict is detected (changes of User 2 and User 3 conflict) → both versions are presented	→ save succeeded → new version is published
4	reinserts his content	reinserts his content	
5	SAVE	SAVE	
6	→ save is aborted → conflict is detected (changes of User 1 and User 2 conflict) → both versions are presented	→ save succeeded → new version is published	
7	manually merges both changes		
8	SAVE		
9	→ save succeeded → new version is published		
10		RELOAD	RELOAD

TABLE I
SUMMARY OF SCENARIO WITH MEDIAWIKI

IV. THE CO-OPERATIVE EDITING SCENARIO IN MEDIAWIKI

In this section we run through the scenario of Section III when using Mediawiki, the system currently used in Wikipedia. We evaluate Mediawiki according to the criteria presented in Section II-B.

A. Revisiting Motivating Example

When users concurrently try to save their changes, the save of only one user succeeds and the other saves fail. The changes of the user whose save succeeds are published. The other users will be presented with two versions of the wiki page: the one that the user tried to publish and the last published version. Conflicts have to be manually resolved by users by re-typing or copying and pasting the changes they performed in the last version that was published. In Table I we illustrated a scenario where User 3, User 2 and User 1 succeed to save their changes in this order. Users solve conflicts by maintaining their modifications in the version to be published and possibly by canceling other concurrent changes. The final result obtained combines the changes of User 1 and User 2, but ignores the modifications performed by User 3.

B. Evaluation

a) Communication Complexity: Generally if we consider m sites and we suppose that when changes are performed in parallel by several users they all try to commit their changes, the total number of messages exchanged is $m(m+1)$. At the beginning all m sites try to save their changes and therefore m messages are sent. One site will receive an accept message, while $m-1$ sites will receive an abort message. Therefore, in this step $2m$ messages are exchanged. In the next step the $m-1$ sites still have to publish their changes. After merging their changes these $m-1$ sites try to save their document versions. One site will succeed the save, while the other $m-2$ will fail. In this step $2(m-1)$ messages are exchanged. The same process continues until the last site saves its version. The total number of messages is $m(m+1)$. After the last save succeeds, the other $m-1$ sites have to reload the latest version. As a reload action requires 2 messages to be exchanged between client and repository, the total number of messages required to ensure convergence of all copies is m^2+3m-2. In the scenario of Table I 16 messages are exchanged.

b) Time Complexity: In Mediawiki systems no merging is performed as users manually combine the parallel modifications. However, differences between document versions are computed to help users in the merging process.

c) Space complexity: Each time a user saves his changes in the repository, the system creates a new version of the document by storing the full content of the updated document version. Therefore the space complexity can be evaluated as the sum of the sizes of all document versions. For instance, in our example there are stored four versions of the document: the base version and three versions created by the three users.

d) First Site Convergence Latency: In the general case where m sites perform concurrent changes and all try to commit at the same time, $2m-1$ rounds are needed before one of the sites converges to the final document state. During the round 1 all m sites try to save their changes. One site will receive an accept message, while $m-1$ sites will receive an abort message. Therefore, in round 2, $m-1$ sites have to perform manual merging. In round 3, these $m-1$ sites try to save their versions. One site will succeed the save, while the other $m-2$ will fail. The same process continues until the last site succeeds to save its version. In our scenario, 5 rounds are needed in order that first user obtains the final state.

e) Convergence Latency: In MediaWiki systems, once a site publishes the final document version, only one additional round is needed in order that all sites receive this version by reloading the new version from the server. Therefore the convergence latency is $2m$ rounds. In our example, 6 rounds are needed before all sites obtain the final document version.

f) Semantic Expressiveness: Since users manually perform merging, changes are not represented by means of operations and therefore no issues concerning semantic expres-

siveness of operations are present. Since a user is in charge of the merging process, the published document version can always be considered as coherent.

g) Determinism: The approach is not deterministic since merging is performed manually and its result is not predictable as it depends on the two versions presented to the user. One of these two versions results from a previous manual merge.

V. P2P COMMUNICATION: GOSSIP PROTOCOLS

Various mechanisms for the propagation of operations among sites can be used. Epidemic propagation is one suitable mechanism for disseminating messages to the whole system. Epidemic or gossip protocols are simple and extremely robust. They can be used to reliably disseminate data in large-scale systems [3]. By analogy with the spread of rumors or epidemics among people, they rely on a continuous exchange of information between peers. They have been applied to a large number of applications. In this paper, we focus on reliable data dissemination to propagate updates among a set of replicas.

A. Properties of Gossip Dissemination

Gossip protocols have been introduced to reliably pass information among a large set of interconnected nodes and turn out to be extremely robust in highly dynamic settings. The gossip protocols robustness relies on the use of randomization to provide probabilistic guarantees and cope with dynamism.

Gossip protocols trade strong deterministic guarantees against probabilistic ones. When applied to gossip-based group communication, we assume that ensuring a reliable dissemination with a high probability is reasonable as long as this probability can be accurately defined. Probabilistic dissemination was first introduced in Pbcast [1] as a back-up mechanism to recover messages lost using IP multicast. It has been shown that if a peer gossips to k other peers chosen uniformly at random among all other peers, the probability that a given peer gets the message is $1 - e^{-\pi k}$ and is actually independent of the system size.

In a later work [6], it has been shown that the probability of achieving an *atomic* broadcast (i.e., all nodes get a message) is $e^{e^{-c}}$ if each node gossips a message once to $k = \log n + c$ other peers chosen uniformly at random, n being the size of the system and c a parameter of the system. This property holds if k is *on average* $O(\log n)$ regardless of the distribution.

These theoretical results can be used to parameterize a dissemination gossip protocol. As pointed out, gossip protocols rely on some form of randomization and redundancy to ensure a reliable dissemination of messages in a large set of nodes in peer to peer systems.

B. Random peer sampling

Peer to peer systems rely on a symmetric communication model where each peer may act both as a client and a server and has a limited knowledge of the system. Therefore, it is unreasonable to consider that each node knows every other node in the system. However, gossip protocols assume that each peer is able to choose uniformly at random a set of f peers to forward a message to. A protocol providing each peer with a random sample of the network is then required.

Although several approaches can be considered, that we can not survey for obvious space reason, to sample a large network, we present here an overview of the peer sampling service [5], a generic substrate to provide each peer with a uniform sample of the network. In this framework, a gossip protocol is executed as follows: Periodically each peer picks a random target from its local view v of the system, exchanges some information with it and processes the received information. If the information exchanged is about the nodes themselves, this protocol builds an unstructured overlay network. The gossip protocol is characterized by the three following parameters:

- **Peer selection:** each peer chooses periodically a gossip target from its current set of neighbors v_i.
- **State exchanged:** the state exchanged between peers is membership information and consists of a list of peers.
- **State processing:** upon receipt of the list, the receiving peer merges the list of peers received with its own list to compose a new list of neighbors.

These parameters can actually be tuned so that the resulting graph exhibits properties which are extremely close to those of a random graph and therefore provides each peer with a uniform random sample of the network that can be used by the dissemination protocol.

VI. MERGE BASED ON OPERATIONAL TRANSFORMATION

A. Presentation of the MOT2 Approach

Operational Transformation (OT) [4] is a well accepted method for consistency maintenance in group editors. It allows each site to execute local operations immediately. Concurrent and non-commutative remote operations, received in arbitrary order, are transformed before execution in order to achieve convergence. MOT2 [2] is an asynchronous merge algorithm using OT that enables divergent replicas to be reconciled pair-wise, at any time, regardless of the pair, while achieving convergence of all replicas. An epidemic membership service can be used for maintaining the group of sites that host replicas. MOT2 is fully symmetric and decentralised, and does not require any external ordering mechanism (such as timestamps or vector clocks). Therefore, the size of the group is not fixed, the peers being able to join and leave the network at any time. The MOT2 algorithm is based on the ability, provided by the use of OT function, called forward transposition [14], to insert a remote operation op inside the history of a site (see Figure 3) without having to undo and redo some operations. For this purpose two conditions are required:

- op is defined from the state resulting from the execution of operations (op_1 to op_{t-1}) located before the insert position t in the history;
- all operations located after the insert position t (sequence seq) are concurrent to op.

Insertion is achieved by a procedure called *Integration*$(H_S, t, \langle S_{op}, op\rangle, op^{seq})$ initially proposed in

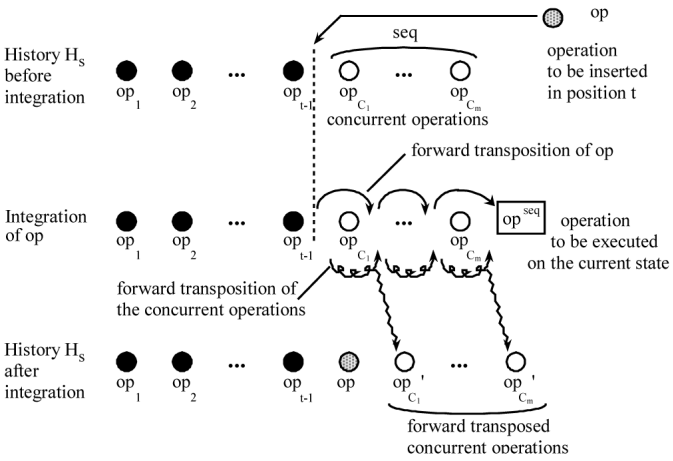

Fig. 3. Integration of an operation in MOT2.

[17]. This procedure receives as an input the history H_S, the insert position t, the operation op to insert and its generator site S_{op}. It delivers as a result the operation noted op^{seq}. Along integration, the operation op is forward transposed with each operation of seq; the resulting operation op^{seq} will be executed on the current state of the replica. During the calculation of op^{seq}, each operation of seq is also transposed to take the insertion of op into H_S into account.

MOT2 assumes that an ordering relation is defined among replicas. This ordering relation is used by MOT2 to serialize concurrent operations according to their generator sites. A unique global order between the operations can thus be dynamically built without requiring a centralizing or ordering mechanism. As a result, histories produced by MOT2 are such that the sequences of operations common to various histories appear in the same order. Consequently the histories which have integrated the same operations are identical.

```
procedure MOT2(S_i , H_j);
// Find the prefix H_C common to H_i and H_j and the index k_S of its last operation
k := k_S + 1;
while (k ≤ sizeofH_i) and (k ≤ sizeofH_j) loop
  ⟨S_op_i , op_i⟩ := H_i(k);
  ⟨S_op_j , op_j⟩ := H_j(k);
  case S_op_i ? S_op_j of
    S_op_j = S_op_i :                    // operation is already present in H_i and H_j
    S_op_j < S_op_i :                    // op_j has to be integrated into H_i and executed
      Integration(H_i, k, ⟨S_op_j , op_j⟩, op_j^seq);
      execute(op_j^seq, R_i);
    S_op_i < S_op_j :                    // op_i has to be integrated into H_j
      Integration(H_j, k, ⟨S_op_i , op_i⟩, op_i^seq);
  endcase;
  k := k + 1;
endloop;                                 // the end of H_i or H_j has been reached
while k ≤ sizeofH_j loop                 // end of H_i :
  ⟨S_op_j , op_j⟩ := H_j(k);
   append(H_i, ⟨S_op_j , op_j⟩);         // append the remainder of H_j to H_i
   execute(op_j , R_i);
   k := k + 1;
endloop;
end MOT2
```

In order to get reconciled, two sites need to transmit their history to each other and then each one has to independently execute MOT2. So MOT2 may be executed by any site S_i. It accepts any history H_j as input, and reconciles its replica R_i with R_j by merging histories H_i and H_j. To simplify we assume that the whole history H_j is available to S_i. The possibility of transmitting to S_i only the part of H_j following the prefix common to H_i and H_j is not presented here.

In MOT2, two operations are considered concurrent when they immediately follow the prefix common to both histories. MOT2 determines first the common prefix. Then, the generator sites of operations that follow the common prefix in H_i and H_j are compared. If the compared operations op_i and op_j have the same generator site ($S_{op_i} = S_{op_j}$), they are identical, meaning that the operation is common to both histories H_i and H_j. When the compared operations op_i and op_j satisfy $S_{op_j} < S_{op_i}$, that means operation op_j is missing in H_i; so it has to be integrated into H_i and executed on the current state of the replica R_i after having been forward transposed with operations following it in H_i. When the compared operations op_i and op_j satisfy $S_{op_i} < S_{op_j}$, that means operation op_i is missing in H_j. It has to be integrated into H_j in order that operations following it in H_j are transposed. Finally the common prefix is augmented by one operation and the process is repeated until the end of one of the histories. The remaining operations of H_j are then appended to H_i.

B. Revisiting Motivating Example

The actions performed by the users are expressed by the operations: $insert(10)$, $update(9)$ and $delete(8-10)$. Operations $insert(10)$ and $update(9)$ commute. Non-commutative operations are transformed as follows:

$TransposeForward(delete(8-10), insert(10)) = null,$
$TransposeForward(insert(10), delete(8-10)) = delete(8-11),$
$TransposeForward(delete(8-10), update(9)) = null,$
$TransposeForward(update(9), delete(8-10)) = delete(8-10).$

An operation (insert or update) concurrent with the delete operation is ignored. Another choice could have been done in order to obtain other effects such as in the WOOT approach described in Section VII. Three reconciliations are needed to obtain convergence of the replicas. The ordering relation is assumed to be: $S_3 < S_2 < S_1$.

After users perform the concurrent changes, a reconciliation between sites S_1 and S_2 is achieved. The resulting histories are: $H_1 = H_2 = update(9); insert(10)$. During reconciliation S_1 executes operation $update(9)$ while S_2 executes $insert(10)$. Then, S_2 and S_3 decide to get reconciled. During reconciliation S_2 has to execute operation $delete(8-11)$ (i.e., $delete(8-10)$ forward transposed with the sequence $update(9); insert(10)$). The resulting histories are: $H_2 = H_3 = delete(8-10); null; null$. Indeed, the transformations of $update(9)$ and $insert(10)$ with respect to $delete(8-10)$ return the $null$ operation. Finally, a reconciliation is achieved either between S_1 and S_2 or between S_1 and S_3. Then S_1 has to execute operation $delete(8-11)$ and the final history for all sites is: $delete(8-10); null; null$. The final history depends on the ordering relation among sites. With the site order $S_1 < S_2 < S_3$, the final history is: $insert(10); update(9); delete(8-11)$. Note that the final replica state is independent of the ordering relation.

C. Evaluation

a) Communication Complexity: The number of messages exchanged between *m* sites depends on the number of reconciliations. A reconciliation between two sites requires two messages for exchanging their histories. The minimal number of reconciliations required in order that the two first sites converge to the final state is $(m-1)$. Then, $(m-2)$ reconciliations are required to make all other sites converge. So, the total number of reconciliations will be $(m-1)+(m-2) = (2m-3)$. Therefore, the total number of messages required to ensure convergence of all replicas is $2(2m-3)$.

b) Time Complexity: Let us consider two histories to be merged : $H_i = H_C.seq_i$ and $H_k = H_C.seq_k$. Assuming that the sequences seq_i and seq_k respectively contain n_i and n_k operations, the number of operations to be integrated by MOT2 is about $(n_i + n_k)$. Besides, integrating an operation op_i (resp. op_k) belonging to seq_i (resp. seq_k) into the history H_k (resp. H_i) results in executing the *Integration* procedure. This procedure has a complexity of $2n_k$ (resp. $2n_i$) due to a double scan of the sequence seq_k (resp. seq_i), first to shift the sequence, then to forward transpose the operations. It results that the MOT2 algorithm executed on each site has a complexity of $O(n_i n_k)$. The total time complexity to obtain the first site convergence, in the worst case, is $O(n^2)$, where $n = \sum_i n_i$ (with $i = 1 \dots m$) is the total number of concurrent operations to be reconciled. The time complexity to propagate this final state is $O(mn)$.

c) Space complexity: Each site S_i manages one replica R_i and the history H_i of operations executed on R_i. Therefore, the space complexity is $O(n)$.

d) First Site Convergence Latency: As previously seen the minimal number of reconciliations required in order that two sites converge towards the final state is $(m-1)$.

e) Convergence Latency: The minimal number of reconciliations required in order that the *m* sites performing concurrent changes converge towards the final state is $(2m-3)$.

f) Semantic Expressiveness: The MOT2 algorithm is independent of the considered operations. Another set of operations could have been chosen with more or less semantics. So, insert, update and delete (concerning several lines) are semantically richer than insert and delete operations used in WOOT approach described in Section VII. The constraints between operations appear when specifying the forward transposition functions. To guarantee replica convergence, the forward transposition functions have to meet condition TP1 [11] which is summed up by state equality: $\forall state\ R_i, R_i \cdot op_1 \cdot op'_2 = R_i \cdot op_2 \cdot op'_1$ with $TransposeForward(op_2, op_1) = op'_1$ and $TransposeForward(op_1, op_2) = op'_2$.

g) Determinism: The MOT2 approach is deterministic. Indeed, the final state of replicas is determined by only considering concurrent operations and the forward transposition functions defined for the application. In particular, the final state of replicas is independent of the order in which sites are reconciled; it is also independent of the ordering relation used among the (generator) sites. However, the final history that is the same on all sites depends on this ordering relation.

VII. WOOTO Approach

In this section we present and evaluate the WOOTO framework [18], an optimised version of the WOOT approach [9], and show how it is applied on the scenario described in Section III.

A. Model

A wiki system based on the WOOTO approach is composed of a set of servers hosting replicated wiki pages. A *wiki server* is a WOOT site with a unique identifier. All WOOT site identifiers are totally ordered. A *wiki page* is identified by a unique identifier $pageid$ assigned when the page is created. A wiki page is composed of a sequence of lines modeled as four-tuples $\langle idl, content, degree, visibility \rangle$:

- idl is the unique identifier of the line represented as the pair $(siteid, clock)$, where $siteid$ is the identifier of the site that created the line and $clock$ is the Lamport clock [7] of the site at the generation time of the line;
- $content$ represents the content of the wiki line;
- the $degree$ of a line is an integer computed by the WOOTO algorithm when the line is generated. Its computation will be explained later on in this section;
- the $visibility$ of a line is represented as a boolean. In the WOOT approach lines are not physically deleted, they are just marked as invisible.

Editing wiki pages is achieved by means of two operations:

- $insert(pageid, line, l_P, l_N)$ inserts a new line $line = \langle idl, content, degree, visibility \rangle$ in a page identified by $pageid$ between the lines identified by l_P and l_N.
- $delete(pageid, idl)$ sets the visibility of the line identified by idl to false in the wiki page identified by $pageid$. Optionally, the content of the deleted line can be garbage collected.

When a new page is created, the page is initialized as a sequence of two sentinel lines L_B and L_E indicating the begin and the end of a page, respectively. When site x generates an insert operation on page p, between $lineA$ and $lineB$, it generates the operation $insert(p, \langle (x, {+}{+}clock_x), content, d, \mathsf{true} \rangle, idl(lineA), idl(lineB))$ where $d = max(degree(lineA), degree(lineB)) + 1$. By definition, L_B and L_E have a degree of 0.

B. Algorithm

Every generated operation is disseminated by using epidemic propagation (see section V) to all sites. Sites can dynamically join and leave the group during the collaboration as WOOT approach does not make any assumption on the size and topology of the group. Each generated operation must be integrated on every site including its generation site. The WOOT algorithm is able to integrate operations and to compute the same result independently of the integration order of operations. This independency relies on the fact that the pairs of operations $(insert, delete)$ and $(delete, delete)$ are commutative. The pair $(insert, insert)$ does not commute

naturally, but the WOOTO data structure enables them to commute. When an insert operation $insert(pageid, line, l_P, l_N)$ is received at a site, lines might be present between the lines l_P and l_N. If no line exists between the lines l_P and l_N, it means that the context of execution of an operation has not changed since its generation. So the new line can be safely inserted between l_P and l_N. In the case that some lines are present between l_P and l_N, the exact insertion position has to be determined. Sorting lines according to their identifiers is not an adequate solution, since the order of already-inserted lines cannot be changed. It is worthwhile to point out that these already inserted lines might not be ordered according to their identifiers [9].

However, if two lines are concurrently inserted between two given lines, the order between the concurrently inserted lines can be arbitrary determined according to the line identifiers. The solution we adopted was to take into account the causal order of insertion of lines and to consider an arbitrary order for lines inserted concurrently such as the order of line identifiers. The degree of lines expresses information about the causal order of line insertions. When a line has to be inserted between two other lines l_P and l_N, the lines with the minimum degree between l_P and l_N are considered first. The position of insertion of the new line is determined according to the ascending order of line identifiers. This represents the position of insertion in the range of lines with the same degree. Lines with other degrees have to be considered as well and therefore the procedure is recursively called for the insertion of the line between the determined position and the right next position. The *IntegrateIns* procedure for determining the position of insertion of a line is presented below. The line to be inserted as well as the line identifiers between which insertion has to be performed are provided as arguments of the procedure. S denotes the sequence of lines composing the page where the new line has to be inserted.

```
IntegrateIns(l, l_P, l_N) :−
  let S' := subseq(S, l_P, l_N);
  if S' := ∅ then
    insert(S, l, position(l_N));
  else
    let i := 0, d_min := min(degree, S');
    let F := filter(S', λl_i. degree(l_i) = d_min);
    while (i < |F| − 1) and (F|i| <_id l) do i := i + 1;
    IntegrateIns(l, F[i − 1], F[i]);
  endif;
```

C. Revisiting Motivating Example

In what follows we describe how WOOTO algorithm can be applied on the motivating example presented in section III. In the WOOTO algorithm, wiki pages and lines are uniquely identified. We suppose that lines 8, 9, 10 were respectively inserted by the 8th to 10th operations generated by site 1 on page 0. Their identifiers are therefore (1,8), (1,9) and (1,10) respectively. We suppose also that lines were inserted in their order of occurrence in the page, and therefore the degree of line is the line number, e.g. degree of line 8 is 8. Consequently, the operation generated at site 1 is represented as: $op_1 = insert(0, \langle(1, 15), \text{“Usually [...]”}, 11, \mathsf{true}\rangle, (1, 9), (1, 10))$

We supposed that the identifier of the new line inserted by op_1 is $(1, 15)$. As this line was inserted between the two lines with degree 9 and 10, the degree of the new line is computed as $max(9, 10) + 1 = 11$.

Update operations are not directly represented in WOOT. An update of a line is interpreted as a delete of the line followed by an insert of the modified line. Thus, the operation generated at site 2 is represented as: $op_{21} = delete(0, (1, 9))$; $op_{22} = insert(0, \langle(2, 7), \text{“[...] Coda [...]”}, 11, \mathsf{true}\rangle, (1, 9), (1, 10))$.

Deletion of multiple lines is not directly represented in WOOT. An operation of deletion of multiple lines is simulated as a sequence of deletions of each line. The operation generated at site 3 is decomposed as the following sequence of operations: $op_{31} = delete(0, (1, 8))$; $op_{32} = delete(0, (1, 9))$; $op_{33} = delete(0, (1, 10))$.

Applying in any order the set of operations op_1, op_{21}, op_{22}, op_{31}, op_{32} and op_{33} leads to the following result at all sites:

$\langle(1, 8), \text{“With no coordination [...]”}, 8, \mathsf{false}\rangle$
$\langle(1, 9), \text{“Conflicts happen when operations fail [...]”}, 9, \mathsf{false}\rangle$
$\langle(2, 7), \text{“[...] Coda [...]”}, 11, \mathsf{true}\rangle$
$\langle(1, 15), \text{“Usually [...]”}, 11, \mathsf{true}\rangle$
$\langle(1, 10), \text{“Resolution of conflicts [...]”}, 10, \mathsf{false}\rangle$

The effects of all operations have been integrated. All lines deleted by site 3 are marked invisible. The new line *“Usually, detection of conflicts is done by using the happens-before relationship in order to flag conflicts.”* created by site 1 is included in the final page. The line modified by site 2 is updated with its new content.

D. Evaluation

a) Communication Complexity: We consider that m sites execute in parallel a set of operations and the operations executed by each site are grouped and sent as a single message. If the diffusion protocol is based on multicast, the communication complexity equals to m messages. If unicast mechanisms are used for diffusion, the total number of messages is $m(m-1)$. In the example presented in Section III, if multicast mechanisms are used, the total number of messages exchanged is 3 (one message per site). If unicast mechanisms are used, the number of messages exchanged is 6.

b) Time Complexity: In the worst case, the time complexity for integrating an operation in WOOTO is $O(l^2)$ [18] where l is the number of lines ever inserted in the wiki page. Therefore, the time complexity for integrating n operations is $O(nl^2)$.

c) Space complexity: Lines are not physically deleted in WOOT approach, but just marked as invisible. Therefore, the space complexity of the approach is proportional to the number of lines ever inserted in the document.

d) First Site Convergence Latency: As the convergence state does not depend on the order of arrival of operations, one round of communication when sites send and receive all messages is sufficient for the first site to obtain convergence.

e) Convergence Latency: One round of communication between sites is sufficient for all sites to obtain the convergence state. Therefore, in WOOT, Convergence Latency is the same as First Site Convergence Latency.

f) Semantic Expressiveness: In the WOOT approach, a shared document is modeled as a linear structure, e.g., a wiki page is represented as a sequence of lines. Only basic operations of insertion and deletion of lines are used to manage the linear structure. Therefore, semantically rich operations are simulated by means of these two basic operations.

The insertion of a line l_i is specified to be performed between two lines l_P and l_N. The precondition to its execution is that these two lines exist. As a postcondition, WOOT approach ensures that the partial orders between lines ($l_P < l_i < l_N$) are maintained as well as previously established orders between other lines.

The deletion of a line l_i marks this line as invisible. The precondition to its execution is that the line exists without taking into account its current visibility status. As a postcondition, WOOT approach ensures that the line is marked as invisible and the previously established orders between lines are maintained.

g) Determinism: WOOT approach is deterministic as the result of convergence does not depend on the order of arrival of operations.

VIII. Action-Constraint Framework

The Action-Constraint Framework (ACF) is a set of tools to reason about consistency in a distributed system [13]. In this section, we focus on the design of a collaborative text editor with ACF, and its evaluation in the context of the generic reconciliation algorithm.

A. Modeling collaborative text editing with ACF

The key concept in ACF is the *multilog*. A multilog is a record of operations submitted by users (actions) and of semantic relations between actions (constraints). A multilog is a tuple $M = (K, \rightarrow, \lhd, \nVdash)$, representing three graphs, where K is a common set of vertices representing actions, and $\rightarrow$, $\lhd$ and $\nVdash$ (pronounced NotAfter, Enables and NonCommuting respectively) are the respective edge sets, called *constraints*. We will explain the semantics of constraints shortly.

ACF is independent of a particular application. Each application defines its own action types, and parameterises the system with constraints between them. For the wiki application, we model the document as a totally ordered set of lines: $D = (L, <_D)$, and each line in L is uniquely identified. Users update D using the following actions:

- $create(c, k, l)$: create a line with content c between lines k and l; return its unique identifier.
- $delete(l)$: hide line l.
- $update(l, c)$: replace the contents of line l with c.

We identify the state of a document with a *schedule*. A schedule S is a sequence of distinct actions, ordered by $<_S$, and executed from the common initial state INIT. In the motivating example, INIT is the state in Figure 1, before users start editing the document. As a simplification, we identify line creation actions by the identifier of the created line. Thus, we note $l_8 = create(\text{“With no ...”}, -, -)$ the action that created line 8; similarly $l_9 = create(\text{“Conflict happen ...”}, -, -)$ for line 9, and $l_{10} = create(\text{“Resolution ...”}, -, -)$ for line 10.

Constraints represent scheduling relations between actions. Basically, $\alpha \rightarrow \beta$ (NotAfter) means that any correct schedule that executes both actions α and β executes α before β. Similarly, if $\alpha \lhd \beta$ (Enables) then any correct schedule that executes α executes β as-well. The following safety condition defines formally the semantics of NotAfter and Enables in relation to schedules. Schedule $S = (A, <_S)$, where $<_S$ is a strict total order over A, is *sound* with respect to multilog $M = (K, \rightarrow, \lhd, \nVdash)$ iff:

$$\begin{cases} \text{INIT} \in A \\ A \subseteq K \\ \alpha \in A \wedge \alpha \neq \text{INIT} \Rightarrow \text{INIT} <_S \alpha \\ (\alpha \rightarrow \beta) \in M \wedge \alpha, \beta \in A \Rightarrow \alpha <_S \beta \\ (\alpha \lhd \beta) \in M \Rightarrow (\beta \in A \Rightarrow \alpha \in A) \end{cases}$$

In the example, assuming that User 3 wants to atomically delete lines 8 to 10, his change is modeled as:

- A set of three actions: $d_1 = delete(l_8)$, $d_2 = delete(l_9)$, $d_3 = delete(l_{10})$.
- An $\lhd$-cycle: $d_1 \lhd d_2 \lhd d_3 \lhd d_1$. Therefore all three delete operations execute, or none of them does.

In our approach, each site i holds its own multilog M_i and its own schedule S_i. M_i monotonically grows over time, by addition of new actions and constraints, either submitted locally, or received epidemically from remote sites (see Section V). S_i represents the current state of the document at site i.

Action $u_1 = update(\text{“Conflict happens ... and Coda ...”},)$ is User 2's update. Notice that actions u_1 and d_1 conflict. Depending on the desired effect, the application may declare them, either to be non-commuting: $u_1 \nVdash d_1$, or antagonistic: $u_1 \leftharpoonup\!\!\rightharpoondown d_1$ (shortcut for $u_1 \rightarrow d_1 \wedge d_1 \rightarrow u_1$). In what follows, we declare them to be antagonistic.

The system has the obligation to eventually resolve conflicts. In the case of non-commuting actions, it must either order them (by adding a NotAfter constraint), or abort one or the other or both. In the case of an antagonism, it can only abort. We explain later how sites agree on a final common state.

Designing an application in ACF is invariant-driven. Let us consider the following (informal) set of invariants:

1) Any two collaborators eventually observe any two visible lines in the same order.
2) If line l is created, eventually all collaborators see l, or none of them.
3) Given a line l, l has eventually the same content for all collaborators.

Tables II and III specify constraints that satisfy these invariants. ($k \leq_D k' \triangleq k <_D k' \vee k = k'$, $o' \prec o$ means that o happens-before o', $o' \parallel o$ means that o and o' are concurrent: $o' \not\prec o \wedge o \not\prec o'$, and $l = o'$ means that the identifier of l (a line) and that of o' (an action of creation) are equal.)

B. Agreeing when collaborating

The system propagates the contents of multilogs, using background epidemic communication. Eventually, users be-

o \ o'	$create(c', k', l')$	$delete(l')$	$update(c', l')$
$create(c, k, l)$	$o' \rightarrow o$ $\left.\begin{array}{l} k = o' \\ \vee\, l = o' \end{array}\right\} \Rightarrow o' \lhd o$	Ø	Ø
$delete(l)$	$l = o' \Rightarrow o' \overset{\lhd}{\rightarrow} o$	Ø	Ø
$update(c, l)$	$l = o' \Rightarrow o' \overset{\lhd}{\rightarrow} o$	Ø	$(l = o')$ $\Rightarrow o' \rightarrow o$

TABLE II
COLLABORATIVE EDITING CONSTRAINTS FOR $o' \prec o$

o \ o'	$create(c', k', l')$	$delete(l')$	$update(c', l')$
$create(c, k, l)$	$\left.\begin{array}{l} k \leq_D k' <_D l \\ \vee\, k' <_D k <_D l' \end{array}\right\} \Rightarrow o \between o'$	Ø	Ø
$delete(l)$	Ø	Ø	$l = l'$ $\Rightarrow l \overset{\leftarrow}{\rightarrow} l'$
$update(c, l)$	Ø	$l = l'$ $\Rightarrow l \overset{\leftarrow}{\rightarrow} l'$	$(l = l')$ $\Rightarrow o' \between o$

TABLE III
COLLABORATIVE EDITING CONSTRAINTS FOR $o' \parallel o$

come aware of the actions of their collaborators, and the possible conflicts. Thus every user eventually receives the insertion: $l_{11} = create$("Usually, the detection... ", l_9, l_{10}) from User 1, the update: u_1 from User 2, and the actions d_1, d_2, d_3 with the parcel constraint from User 3. According to Table III, every site eventually includes the following information in its multilog: the set of actions $\{l_{11}, u_1, d_1, d_2, d_3\}$, the parcel constraint: $d_1 \lhd d_2 \lhd d_3 \lhd d_1$, and the conflict: $u_1 \overset{\leftarrow}{\rightarrow} d_2$.

However, until operations commit, different users may view different states of the document. For instance if User 1 ignores User 3, his current view could be the following sound schedule: $S_1 = \text{INIT}.l_{11}.u_1$. Similarly User 3 might observe his own actions only: $S_3 = \text{INIT}.d_1.d_2.d_3$.

Eventually, sites have to agree. We propose a voting protocol, whereby each site makes a proposal reflecting its tentative state and/or the user's preference [15].

Back to the example. Actions u_1 and d_1 are antagonistic (linked by a NotAfter cycle), and actions d_1, d_2 and d_3 are atomic (linked by an Enables cycle). Say User 1 and User 2 both vote to commit l_{11} and u_1; consistency obligates them to vote to abort d_1, d_2 and d_3. Note this proposal P, and note Q the proposal of User 3 to commit his own actions and to abort both l_{11} and u_1. Our protocol distributes proposals epidemically. Eventually all sites are aware of P and Q.

Our algorithm decomposes a proposal into semantically-meaningful units, called candidates. An election runs locally between competing candidates. A candidate C receives a number of votes, equal to the sum of the weights of the sites that voted for some proposal containing C. (If a single site has a weight of 100%, our algorithm degenerates to a primary approach.) Suppose that weights are uniformly distributed among three sites. Then candidate $C =$ "commit u_1" extracted from P has a weight of $\frac{2}{3}$, and candidate $C' =$ "abort u_1" extracted from Q has a weight of $\frac{1}{3}$.

A candidate cannot be just any subset of a proposal. It must also be consistent with existing constraints. For instance "abort d_1" is not a well-formed candidate, because of the atomicity constraint, but "abort d_1 and d_2 and d_3" is well-formed. During an election a candidate competes against comparable candidates. Two candidates are comparable if they contain the same set of actions. For instance candidates C and C' are comparable. A candidate wins when it receives a majority or a plurality. In our example, eventually every site aborts d_1, d_2 and d_3, and commits l_{11} and u_1.

C. Evaluation

a) Communication Complexity: We consider that m sites execute an action concurrently. Sites exchange their proposals and their multilogs epidemically. In the best case, the communication cost is $4(m-1)$:

- Every site sends its actions to site i: $m-1$ messages.
- Site i computes the constraints, and sends its multilog M to all other sites: $m-1$ messages.
- When a site receives M, it computes a proposal and returns the result to i: $m-1$ messages.
- Site i receives all proposals, and sends them to other sites: $m-1$ messages.
- Each site decides locally.

This reduces to $2(m-1)$ if a single site holds 100% weight.

b) Time Complexity: Computing an optimal proposal is equivalent to the feedback vertex set problem, which is NP-hard. However, the IceCube system proposed heuristic algorithm, which computes an excellent approximation of the optimal with $O(n)$ average complexity, where n is the number of actions in the multilog [10]. However, the complexity of the election algorithm is $O(m^3n^2)$ in the worst case [15], which dominates the cost of computing a proposal.

c) Space complexity: In ACF a site stores all the non-stable actions, be they either local or remote. However, stable actions and their constraints are eventually garbage-collected, and replaced by snapshots. A snapshot contains the state of the document, its size is proportional to l. If we assume that GC keeps a small and constant number of snapshots at each site, the space complexity is $O(lm)$.

The size of a proposal is eventually $O(n)$. As each site keeps tracks of all proposals, the space requirement for proposals is $O(nm)$. (Proposals are also eventually garbage-collected, but we ignore this effect in this evaluation.)

d) First Site Convergence Latency: In the best case execution (the one depicted above), the number of asynchronous rounds to converge is 3. It reduces to 1 if a primary site holds all the weight.

e) Convergence Latency: Once a site has elected a candidate locally, a single additional round ensures other sites are informed.

f) Semantic Expressiveness: ACF supports arbitrary operation types. The ACF logic is parameterised by application semantics; see for instance Tables II and III. A different application differs only by a different set of constraints.

Conflict is an important concept in collaborative applications, and ACF supports it. ACF recognises two variants of conflict, non-commutativity and antagonism. ACF also allows

Comparison criteria	MediaWiki	MOT2	WOOTO	ACF
Merging concurrent changes	Manual	Automatic	Automatic	Automatic
Communication topology	Centralised	Decentralised (Pair-wise synchro.)	Decentralised	Decentralised
Dynamic membership	N/A	Yes	Yes	No
Communication complexity	$m^2 + 3m - 2$	$2(2m - 3)$	m	$4(m - 1)$
Time complexity	N/A	$O(n^2 + mn)$	$O(nl^2)$	$O(m^3n^2)$
Space complexity per site	$O(L)$	$O(n)$	$O(l)$	$O(l + mn)$
First site convergence latency	$2m - 1$ rounds	$m - 1$ rounds	1 round	3 rounds
Convergence latency	$2m$ rounds	$2m - 3$ rounds	1 round	4 rounds
Semantic expressiveness	N/A	Any operation	Insert, Delete	Any operation + constraints
Deterministic	No	Yes	Yes	No / Yes (given votes)

Key: m = number of sites, n = number of operations, L = number of lines in the doc., l = number of lines ever appeared in the doc.

TABLE IV
COMPARISON OVERVIEW

users to group operations atomically. This is particularly useful in our example. For instance, if a user inserted a line between lines 8 and 9, it probably would not make sense to keep the inserted line without the lines around it. In our system, this will be flagged as an antagonism, and either the insert or the delete would fail (or both). In contrast, the WOOTO approach is limited to commutative operations, and MOT2 to ones that can be transformed to commute. These systems do not support conflict nor atomicity. In fact, these techniques are complementary. MOT2 or WOOTO should be used to make commutative as many operation pairs as possible; then the ACF reconciliation techniques can be used to resolve any remaining conflicts.

g) Determinism: By design, our approach is not deterministic, since the outcome depends on the collaborators' votes. However, for a given set of votes, the system is deterministic.

IX. CONCLUSION

In this paper we presented four approaches to collaborative editing of wiki pages: the current centralised approach, and three decentralised, peer-to-peer approaches. One is based on operational transformation, one on commutative operations, and one on reconciliation. We discussed and evaluated each one in detail, according to a complete set of metrics. Table IV summarises the evaluation.

None of the three proposed approaches is better than the others. In terms of expressiveness ACF is better than MOT2 which at its turn is more general than the specific solution proposed by WOOT. Regarding convergence speed in terms of rounds and message traffic required, WOOT and ACF are the most efficient approaches. Finally, MOT2 and WOOT are adapted for dynamic membership required in any P2P network, while ACF requires a static membership

REFERENCES

[1] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal Multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[2] M. Cart and J. Ferrié. Asynchronous Reconciliation based on Operational Transformation for P2P Collaborative Environments. In *Proc. of the International Conference on Collaborative Computing - CollaborateCom 2007*, White Plains, New York, USA, Nov. 2007.

[3] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proc. of the 6th Annual ACM Symposium on Principles of Distributed Computing - PODC'87*, pages 1–12, Vancouver, British Columbia, Canada, Aug. 1987. ACM Press.

[4] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *Proc. of the ACM SIGMOD Conference on the Management of Data - SIGMOD'89*, pages 399–407, Portland, Oregon, USA, May 1989.

[5] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. The Peer Sampling Service: Experimental Evaluation of Unstructured Gossip-based Implementations. In *Proc. of the 5th ACM/IFIP/USENIX international conference on Middleware - Middleware 2004*, pages 79–98, Toronto, Canada, 2004. Springer-Verlag.

[6] A.-M. Kermarrec, L. Massoulié, and A. J. Ganesh. Probablistic Reliable Dissemination in Large-Scale Systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3), Mar. 2003.

[7] L. Lamport. Times, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[8] N. A. Lynch. *Distributed Algorithms*. Morgan-Kaufman, 1996.

[9] G. Oster, P. Urso, P. Molli, and A. Imine. Data Consistency for P2P Collaborative Editing. In *Proc. of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, Banff, Alberta, Canada, Nov. 2006. ACM Press.

[10] N. Preguiça, M. Shapiro, and C. Matheson. Semantics-based Reconciliation for Collaborative and Mobile Environments. In *Proc. of the International Conference on Cooperative Information Systems - CoopIS 2003*, volume 2888 of *Lecture Notes in Computer Science*, pages 38–55, Catania, Sicily, Italy, Nov. 2003. Springer-Verlag.

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

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

[13] M. Shapiro and K. Bhargavan. The Actions-Constraints approach to replication: Definitions and proofs. Technical Report MSR-TR-2004-14, Microsoft Research, Mar. 2004.

[14] M. Suleiman, M. Cart, and J. Ferrié. Concurrent Operations in a Distributed and Mobile Collaborative Environment. In *Proc. of the International Conference on Data Engineering - ICDE'98*, pages 36–45, Orlando, Florida, USA, Feb. 1998. IEEE Computer Society.

[15] P. Sutra, J. Barreto, and M. Shapiro. Decentralised Commitment for Optimistic Semantic Replication. In *Proc. of the International Conference on Cooperative Information Systems - CoopIS 2007*, Vilamoura, Algarve, Portugal, Nov. 2007.

[16] R. H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[17] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies Convergence in a Distributed Real-Time Collaborative Environment. In *Proc. of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2000*, pages 171–180, Philadelphia, Pennsylvania, USA, Dec. 2000.

[18] S. Weiss, P. Urso, and P. Molli. Wooki: a P2P Wiki-based Collaborative Writing Tool. In *Proc. of the International Conference on Web Information Systems Engineering - WISE 2007*, Nancy, France, Dec. 2007. Springer.