A Multi-version Algorithm for Cooperative Edition of Hierarchically-Structured Documents

Josep M. Ribó
Universitat de Lleida(UdL)
C. Jaume II, 69
E-25001 Lleida (Catalunya, Spain)
josepma@eup.udl.es

Xavier Franch
Universitat Politècnica de Catalunya (UPC),
c/ Jordi Girona 1-3 (Campus Nord, C6)
E-08034 Barcelona (Catalunya, Spain)
franch@lsi.upc.es

Abstract

Several approaches do exist to solve the problem of editing a text document cooperatively in real time. We believe that those approaches could be improved in two ways: (1) preserving the intention of the participants in a better manner and (2) benefiting from a hierarchical document structure (as in XML). This article presents an algorithm for cooperative edition of documents that (1) achieves intention preservation by keeping different versions of the concurrently accessed document fragments and that (2) reduces concurrency conflicts by restricting them to hierarchically dependent fragments.

1. Introduction

The cooperative edition of a text document consists in the edition of a document simultaneously by several users (sites), possibly located in geographically different places, in such a way that any site can see the actions of the others in real time.

Clearly, several concurrency problems arise when such an edition system is to be designed. [13] establishes the three properties that any cooperatively edited document should keep in order to be considered free of concurrency problems: *convergence, causality preservation* and *intention preservation*. These properties mean the following :

Convergence: When the same set of operations have been executed at all sites, all copies of the shared documents are identical.

Causality preservation: For any pair op_1, op_2 of operations such that op_2 depends on op_1 (i.e. op_2 was performed having seen the result of op_1), then op_1 is executed before op_2 at all sites.

Intention preservation: For any operation *op*, the effects of executing *op* at all sites is the same as the intention of the site that executed *op* locally, and the effect of executing *op* does not change the effects of independent operations (i.e. each one of them has not seen the effect of the others).

This article focuses on presenting an algorithm for cooperative edition of documents within a client-server paradigm that keeps the above-mention properties. Specifically, intention preservation is achieved by keeping different versions of the concurrently accessed document fragments. In addition, the algorithm is able to deal with a hierarchical document structure.

This article does not address the general problem of semantic consistency in a cooperative editing system (i.e. meaning of the document, crossed references...). As [13] points out, the maintenance of semantic consistency cannot be solved without the intervention of the people in collaboration. We are not aware of any cooperative editing system that attempts to solve this problem.

Section 2 overviews other approaches to the same problem, and states the features of our solution. Section 3 provides the necessary framework to understand the cooperative algorithm given later. Section 4 defines precisely the consistency model of our solution. Section 5 is the core of the article since it presents the algorithm for cooperative edition of a text document. Finally, section 6 shows a detailed example.

2. Related work and features of our approach

In the last decade, several approaches have been taken in order to solve the problem of the cooperative edition of a text document by keeping the above-mentioned properties. Some of those approaches were based on order–taking, serialization, locking or causal-ordering [2, 4, 6, 14...] but either they did not succeed in achieving all the three correctness properties or they presented some concurrency problems (see [13] for an overview).

[1] took a different perspective and introduced the idea of *operation transformations*, according to which, a set of concurrent (and hence, possibly independent) operations can be properly transformed before their execution so that they lead to the same final document state in all sites, regardless the order in which they have been executed.

This idea has been widely exploited by [9, 10, 12, 13] in order to keep intention preservation (which is the most difficult property to achieve). Although they have used

0-7695-1351-4/01 $10.00 © 2001 IEEE

different strategies, they have got similar results. However, we strongly believe that their approaches suffer from two important problems:

- They do not actually achieve intention preservation. Let us consider the following example: a document contains the word 'pace' and two sites modify it concurrently: site 1 inserts 'e' at position 2 (meaning 'peace'); site 2 inserts 's' at position 5 (meaning 'paces'). Clearly, they have two different (and irreconcilable) intentions. The final result, according to the mentioned approaches will be 'peaces' which does not conform with either of the two intentions. It seems clear that a better solution would involve creating two different versions of this word in the document: 'peace' and 'paces'.
- They do not provide a hierarchical view of the document, which has, at least, two drawbacks: (1) Any pair of independent operations that are performed at any part of the document collide (i.e. the *conflict unit* encompasses the whole document), even if those operations were performed at very distant parts of the document; and (2) they do not provide a direct way to perform operations on non-atomic parts of the document (i.e. copy of a whole paragraph).

Some approaches do exist that create multiple versions of objects in the framework of drawing environments: [7, 11]. However, they have some drawbacks: [7] fails to achieve causality preservation. [11] allows the application of *compatible operations* on the same object without generating a new version, but in our opinion, this leads to problems in preserving intention. Let us consider the following example: two incompatible operations (op_1, op_2) performed on the same object *obj* have generated two different versions of it (obj_1, obj_2). After that, a third operation (op_3) performed on *obj* is brought up by a site that has seen the effects of both op_1 and op_2. Consequently, the intention of this site was to apply op_3 either to obj_1 or to obj_2 (not to both of them!). However, according to [11]'s policy, both obj_1 and obj_2 will be updated with op_3.

We are not aware of the existence of any multi-version approach in the cooperative text edition field.

Neither [7] nor [11] use hierarchical document structure. This issue is addressed in [5], although their solution is quite restrictive since only two sites can access concurrently to the same node and only operations to insert/delete characters are considered (no operations are defined on higher document levels).

Features of our approach

Our approach tries to present a proposal to improve some of the above-mentioned limitations. Specifically, it is based on the following features:

1. Provision of a quick responsiveness to local site's operations (operations are executed locally at once).
2. Sites may reside in geographically different places.
3. Any site can perform operations on the document at any time. All the operations are processed and take effect into the document. (no locking policy. No discarded operations).
4. Ability to deal with a hierarchical document structure (sections, subsections, paragraphs, words...).
5. Convergence and causality preservation are kept.
6. Intention preservation is kept by means of a multi-version approach.
7. Use of a server that processes and broadcasts operations generated by the sites.

Features 4 and 6 are the most important ones since, to our knowledge, they have not been addressed before in the field of cooperative text edition or have been addressed only partially.

The hierarchical document structure allows us to have smaller conflict units in processing concurrent operations, as in [5] (i.e. operations are not performed on the whole document but on a specific document fragment. In the case of [5], this fragment is a word). In addition, our approach can deal with multiple-level conflict units (i.e. we provide a set of operations that may be applied, without restriction, to different levels of the document structure, like a word or a paragraph). This makes it more natural to perform operations that are applied to non-atomic document fragments.

Concerning our multi-version approach, it has two important advantages:

- It preserves sites' intentions in a better way, by ensuring that each operation will be performed at all sites on a document fragment identical to the one on which it was intended originally. Hence, during the generation of a document, and before it gets into a coherent state, various different *conditional* versions of a document fragment may coexist. In the end, some sort of negotiation between sites will be necessary so that only one version of each fragment remains.
- Users will be able to decide on which existing version of a document fragment they want to perform their operations, allowing them to incorporate information to the document that will only make sense if, eventually, that version remains in the document.

Our approach to cooperative editing is a part of a more general project in the framework of Software Process Modelling called PROMENADE ([15, 16]). Within this project, the existence of a server that centralizes site's operations is natural and even necessary (shared documents are kept in a common reservoir).

3. Solution framework

In this section, we present some aspects concerning the solution framework (document structure and operations allowed to sites) and we outline the solution itself.

3.1 Document structure

We take the XML notion of document. In XML, document structure can be defined by means of a *schema* written using either DTDs or XML Schemas [3].

Schemas usually present the structure of a document in a hierarchical way (e.g. a document is made out of sections. Each section has subsections and so on).

Using this view, a specific document may be seen as a tree. Each level represents an element of the hierarchy (e.g. a section). The nodes in a level represent all the instances contained by the document at that level (i.e. a node is an instance of a section, a subsection, a paragraph, a word...).

A node may have different versions. A new version of a node is created by the server when a conflict occurs (see below). Since different versions of the same node may coexist in a particular document, we organize documents more appropriately as trees of *pairs <node_number, version_number>*. All pairs that contain an instance of the same node will have a common *node_number* (and a different *version_number*). Two identical pairs cannot coexist in the same document.

Two attributes are relevant for a pair *<node,version>*:

The site that has generated the conflicting operation which has resulted in the creation of the new version and

whether that version is deleted[1] or not.

Notice that the creation of a new version of a node involves creating a new version of all its subnodes. Figure 1 shows an example of document with a hierarchical structure.

3.2 Operations

Unlike other approaches (and due to our hierarchical approach), the operations allowed by the algorithm of cooperative edition are applied to document nodes (more specifically, to document *pairs*). The following set of operations are considered:

- *insertCharacter(<n,v>,p,c)* and *deleteCharacter(<n,v>,p)*. *insertCharacter* inserts the character c in the position p of a pair *<n,v>* that represents a word, while *deleteCharacter* removes the p^{th} character of the word identified by *<n,v>*. Except for the pair parameter, these are the usual operations offered by many cooperative algorithms ([5, 10, 13...]).
- *insertNode(x,loc)* and *removeNode(<n,v>)*. These operations may be applied to any kind of node (not only to words). Therefore, they allow us to work at any level of the document. Specifically, *insertNode* inserts a new node (which is given a distinct *node_number* and *version_number*=1) with contents x at the location *loc* of the document and *removeNode* removes the pair identified by *<n,v>* from the document. Some specific examples of these operations may be the removal of a paragraph or a section; the move of a paragraph (i.e. cut-and-paste, which will involve the use of both remove and insert operations); the copy of a section to some other place (i.e. copy-and-paste).

Because of the hierarchical organization of the document, an operation may collide with others that have been carried out on different (but dependent) pairs. For instance, at the same time that a site u_1 modifies a word of the document, another site u_2 may remove the whole paragraph in which that word is located. However, if two operations are applied to independent nodes (e.g. two different paragraphs or two different words within the same paragraph) they do not collide.

3.3 Overview of the solution

The solution we propose to the cooperative edition of a text document consists globally in the following:

1. The participant sites perform their operations locally as soon as they generate them (to get a quick responsiveness).
2. After that, they send a request to the server with the operation they have performed along with the last global state that they have received from the server. This request is queued in the server.
3. The server maintains a sequence of the global states that the document has gone through. Queued requests are processed sequentially. Each request is transformed in such a way that the operation it contains can be applied to the last global state of the server maintaining the intention of the site that has generated it. This transformation may involve creating new versions of some nodes in order to avoid conflicts.
4. The server broadcasts the transformed requests to all the sites which apply them locally. In this way we assure that all the sites see exactly the same sequence of operations, hence they converge.

[1] A pair may be ticked as 'deleted' if there has been a conflict between two independent operations (one of which has removed it). In this case, that pair will still appear in the document but conveniently marked as deleted.

The main goal of this article is to present the algorithm performed by the server, which is outlined in step 3. This is done in section 5.

3.4 Example overview

To get an overview of our approach, consider the following example: A document consists of one paragraph (p), which contains two words (w_1="abce" and w_2="efgh". See figure 1(a)). Sites u_1 and u_2 begin to modify the document:

- First of all, they modify w_1 and w_2 respectively (u_1 inserts '1' to w_1; u_2 inserts '2' to w_2). No conflict occurs (concurrent modification of independent nodes. See figure 1(b)).

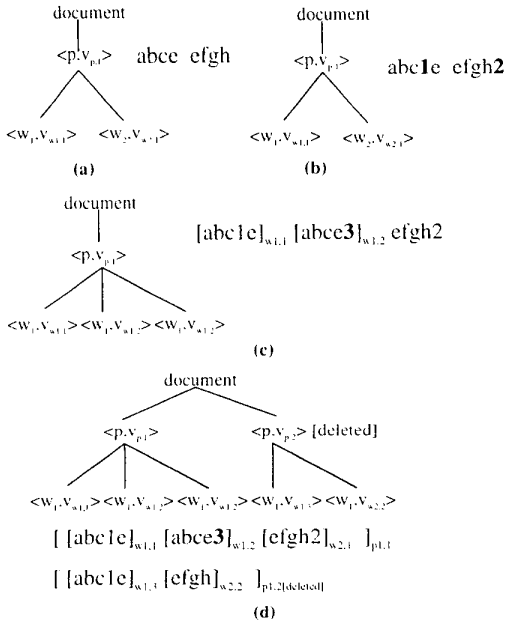

Figure 1. Example overview

- u_2 modifies w_1 without having seen the last u_1's operation to w_1 (u_2 inserts '3' to w_1). A conflict occurs. Another version of w_1 ($<w_1,v_{w1.2}>$) is created with the exact contents that u_2 had seen. u_2's operation is applied to $<w_1, v_{w1.2}>$. See figure 1(c).
- u_1 removes the whole paragraph p (without having seen any of the u_2's operations on w_1 or w_2). A conflict occurs. Another version of p ($<p,v_2>$) is created with exactly the contents that u_1 has erased. This new version is labeled as 'deleted' (figure 1(d)).

This example will be developed in more detail in section 6.

4. Consistency model

As we have said, the consistency of a document which is edited cooperatively by various sites following a policy *What You See Is What I See* is defined by using the properties of *Convergence, Causality preservation* and *Intention preservation.*

We keep the same definitions for the first two properties as those given in [13]. We diverge, however, in the interpretation of *intention preservation.* We have noticed in section 2 that the idea of intention preservation introduced in [13] could lead to unsatisfactory outcomes. In order to avoid those problems we say that a document preserves the intentions of all the participant sites, if each site's operation on the document is performed precisely on the same document node which the site intended to. As an immediate consequence of this, if two sites perform two operations concurrently on the same document element, two different versions of that element are to be created. These two element versions will coexist in the same document.

In order to define precisely our consistency model, the following definitions are given:

Global state. The contents of the document at a given instant as it has been recorded by the server. The server creates a new global state for each operation performed by a site on the document. Therefore, a sequence of global states is kept by the server. A specific global state is identified by an ordered natural number (starting from 0).

The contents of the document at any site k may differ from the last global state S_j received by k (from the server) only in those operations performed by k that the server has not included in S_j (because it has not processed them yet or has processed them in a state posterior to S_j).

Temporal dependency of operations[2]. An operation op_1 (performed by site i at state S_k) is temporally dependent on an operation op_2 (performed by site j at state S_r) if either:

(a) $i=j$ and op_1 has been performed after op_2 or

(b) $i \neq j$ and when i performs op_1 it has already received the effect of op_2.

If neither op_1 is temporally dependent on op_2 nor op_2 is temporally dependent on op_1, we say that op_1 and op_2 are temporally independent operations.

[2] In the remainder of the article we will abbreviate *temporal dependency of operations* to *dependency of operations.*

Dependency of pairs. A pair $<n,v>$ is dependent on another pair $<n_2,v_2>$ if $<n,v>$ belongs to the tree rooted at $<n_2,v_2>$ (in particular, a pair is dependent on itself).

Convergence and causality preservation. We keep the same definitions given in section 1.

Intention preservation. As we have said, the definition of *intention preservation* given in section 1 needs to be adapted to our multi-version approach.

A document conforms with the property of intention preservation if any operation *op* performed by a site *k* on a pair $<n,v>$ at state S_i is performed at all sites on a version of *n* with exactly the same contents that $<n,v>$ had at site *k* at the moment of generating *op*.

As a consequence of this property, an operation op_1 made by a site *k* on a pair $<n,v>$ may lead to a conflict in the case that some other operation op_2 which is independent on op_1, has been already performed on a pair $<n_2,v_2>$ with some dependence with $<n,v>$[3].

Whenever such a conflict is detected, in order to keep the intention preservation property, a new version of the most general of nodes *n* and n_2 will be created with the same contents seen by *k* immediately before op_1 was performed. The operation op_1 will be performed on that new version.

Consistent document. A document which is being edited cooperatively is said to be consistent if it keeps the properties of *convergence, causality preservation* and *intention preservation.*

Coherent document.
A coherent document is a consistent document that has only one version for each node (i.e. all pairs of a coherent document have a different *node_number*).

5. The algorithm

In this section we present the algorithm run by the server to achieve the cooperative edition of a document along with some necessary definitions.

5.1 Sequence of global states and transitions between states

The sequence of global states is created by the server and contains the states through which the document has passed, along with the transitions between those states.

The states are identified by increasing naturals. A transition from the state S_i to the state S_{i+1} is a tuple [*op*, $<n,v>$, *u*], where:

- *op* is the operation to be performed to pass from S_i to S_{i+1}
- $<n,v>$ is the pair on which *op* is applied.
- *u* is the site that has generated *op*.

The last state of the sequence represents the current global state of the document. After generating a state, the server broadcasts the last transition (the one that generates that state) to all the sites.

5.2 Requests

When a site wants to perform an operation, first of all it carries it out locally and then, sends a request to the server in order to be processed, added to the sequence of global states and broadcast to the other sites.

A request is a tuple: [*op*, *u*, S_i, $<n,v>$], where:

- *op* is the operation to be performed,
- *u* is the site performing the operation,
- S_i is the last global state received by *u* from the server before performing *op*.
- $<n,v>$ is the pair on which the operation takes place.

5.3 Conflicting transition

Given a request *req*=[*op*, *u*, S_i, $<n,v>$] and a sequence of global states
$s=\{ S_0, S_1, \ldots, S_i, \ldots, S_k \}$ held by the server,

A transition $t=[op_2,<n_2,v_2>,u_2]$ performed at state S_j is conflicting with *req* if:

(a) $u \neq u_2$

(b) $i \leq j < k$

(c) $<n_2,v_2>$ is dependent on $<n,v>$ or $<n,v>$ is dependent on $<n_2,v_2>$.

That is, an operation op_2 already performed by a site u_2 (different from *u*) at state S_j causes a conflict with the request containing *op* if op_2 has been performed on a pair which is not independent with $<n,v>$ and the result of op_2 has not been seen by *u*.

Given a request *req*=[*op*, *u*, $<n,v>$, S_i] and a sequence of global states
$s=\{ S_0, S_1, \ldots, S_i, \ldots, S_k \}$, *the first conflicting transition with req* is the transition for which (a), (b) and (c) hold and that is performed on the lowest possible global state.

Notice that the hierarchical document structure allows conflicts to be restricted to those operations that occur on dependent pairs. Therefore, unlike other approaches, no time-consuming processing and transformations will be necessary for most operations (which will be applied to independent nodes).

[3] The proper definition of conflict is given at section 5.3.

5.4 Creation of a new version and tree of node versions

The creation of a new version of <*n,v*> (namely, <*n,v'*>) at state S_i as a result of a conflict arised by an operation generated at site *u* consists in creating a copy of the contents of the tree rooted at <*n,v*> at state S_i and inserting it as the next child of <*n,v*> 's parent.
The tree of node versions for a specific *n* keeps track of the history of versions of *n*, along with which site and at which state that version has been generated (see figure 2).

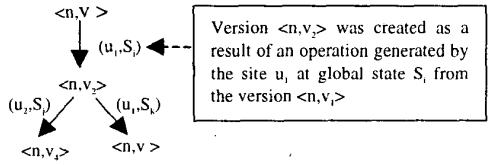

Figure 2. Tree of node versions associated to node *n*

An important property of the tree of node versions is the following: Any node of the tree cannot have two children created by the same site.

5.5 Getting the last node version

In the algorithm that will be presented in next section, it will become necessary to get the last version of a node <*n,v*> that has been created as a result of an operation raised by site *u*.. This last version can be obtained following this sequence:

1. Find the pair <*n,v*> in the tree of node versions associated to *n*
2. Find the only (at most) child of <*n,v*> generated by site *u*. Call it <*n,v'*>
3. Repeat step 2 on <*n,v'*> while there exists such <*n,v'*>

As we have stated in last section, at step 2, there can be at most one child generated as a result of a *u*'s operation.

5.6 Algorithm description

Recall that the overall editing process consists in the following: Each operation on the document is performed locally in the site that has generated it (in order to keep short response times). After that, a request with the operation is sent to the server, which will be in charge of processing and broadcasting it to the other sites.
The algorithm presented in this section describes how the server processes a request from a site while keeping the document consistency (i.e. keeping the properties of *convergence, causality preservation* and *intention preservation*). The achievement of the first two properties is a direct consequence of the approach taken (the server, in fact, serializes the requests and broadcasts them, which make it possible to achieve convergence and causality preservation). Intention preservation is kept by request processing in the server, which is explained in this section.

When a request *req*=[*op, u,* S_p <*n,v*>] reaches the server (which keeps a sequence of global states $s=\{S_0,..., S_p,..., S_k\}$, the objective is to transform that request applied locally at state S_i into a transition that will be applied to the state S_k and will generate S_{k+1}. In order to do so, the following actions are performed:

1. *Find out on which specific version of n, op must be performed:* <*n,v'*>.
   Since *op* is performed locally at site *u*, in order to keep intention preservation, we have to ensure that the server will apply *op* exactly to the same version of *n* as it has been applied locally.
   This version will be *v* in the case that no previous request $req_2=[op_2,u,S_m,<n',v>]$, $(m \leq i)$ has created a new version *v'* of *n* (i.e. *v'* will be created if a conflict comes up when processing req_2, in order to keep intention preservation)[4]. If such a version (*v'*) has been created when processing req_2, it becomes the version of *n* corresponding to <*n,v*> for subsequent *u*'s requests.

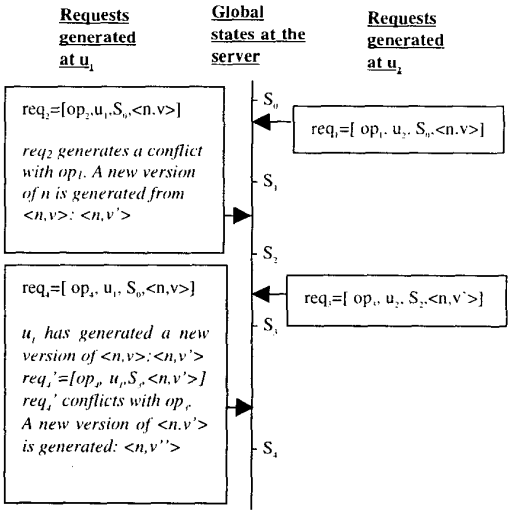

Figure 3. Multiple versions generated by u_1 operations

[4] Notice that *n* and *n'* should not be necessarely the same node. In general, they should be dependent nodes.

In the general case, several conflicts may have occurred when processing requests raised by u leading to the creation of new versions of n (see figure 3). Therefore, the correct version to which *op* should be applied (namely v'), is the last version of n that u has created starting from $<n,v>$ (the algorithm for obtaining v' has been described in section 5.5).

The request *req* will now be transformed into an equivalent request *req'* in which *op* is applied to v': $req'=[op,u,S_r,<n,v'>]$,

S_r is the global state in which $<n,v'>$ was created ($i<r<k$). $S_r=S_i$ in the case that $v'=v$.

2. *Find out if there is any conflicting transition with req'.*

Before applying the request *req'* we have to make sure that the states $S_r..S_k$ do not contain any conflicting transition with it. Recall that a conflicting transition with *req'* will be a transition $t=[op_t,<n_t,v_t>,u_t]$ generated at some state S_j ($r \le j< k$) by a site $u_t \neq u$ on a node n_t such that n_t and n are dependent. If such a transition does exist, this means that $<n,v'>$ may contain some modifications that u had not seen at the moment of generating *op*. Hence, *op* cannot be applied to $<n,v'>$ (if intention is to be preserved) and a new version of the most general node between $<n,v'>$ and $<n_t,v_t>$ has to be created (namely, n_g). The new version of n_g will have the value that it had at state S_j.

We will call t to the first conflicting transition with *req'* encountered within $S_r..S_k$; S_j to the origin state of such a transition and $<n_g, v_g>$ to the pair containing the node (n_g) a new version of which will be created.

If such conflicting transition does not exist, *req'* can be applied straight to S_k in order to generate S_{k+1}.

3. *If the tuple (t,S_j, $<n_g, v_g>$) has been found at step 2, create a new version of $<n_g, v_g>$: $<n_g, v_g'>$*

$<n_g, v_g'>$ will be created with the value of $<n_g, v_g>$ at state S_j. Section 5.4 describes how to proceed to create a new version of a node at a given state.

4. *If a new version $<n_g, v_g'>$ has been created at step 3, apply to it all the u's* remaining *operations*

Subsequent states to S_j may contain transitions performed by u on some other pair of which $<n_g, v_g>$ is an ancestor (see example in section 6). Since these operations were performed locally by u before *op*, they have to be included in the part of the document rooted at $<n_g, v_g'>$ before including *op*.

Hence, in this step, if a new version $<n_g, v_g'>$ has been created in step 3, we apply to it all the operations coming from u, that have been performed at states posterior to S_j on any pair $<m,w>$ such that $<n_g, v_g>$ is an ancestor of $<m,w>$.

5. *Apply op to the right version of n at the state S_k and do the broadcast*

If no conflict has been detected, the application of *op* to $<n,v'>$ (at state S_k) will preserve the intention of u. Hence, generate the transition $t=[op, u, <n,v'>]$ and apply it to S_k in order to generate S_{k+1}.

Otherwise *op* will be applied to the corresponding version of $<n,v'>$ within $<n_g, v_g'>$ (namely $<n,v''>$)[5]. Hence, generate the transition $t=[op, u, <n,v''>]$ and apply it to S_k in order to generate S_{k+1}.

6. *Broadcast t and the new generated version (if any) to all sites.*

5.7 Algorithm for the sites

Although, clearly, the critical algorithm is the one that is executed at the server, some aspects are worth noting about the cooperative editing algorithm that must be run at the different sites. This algorithm includes:

- Updating the local instance of the document with the transitions received from the server and
- Applying the local operations to the local instance of the document before sending them the the server .

It is important to keep at all instants the following property: Two dependent pairs belonging to a local document cannot contain at the same time: (a) an operation *op* that has been performed locally and has not yet been confirmed by the server; and (b) an operation *op'* performed remotely that has been received from the server after the generation of *op*.

This conflict must be resolved by generating a new local version of the corresponding node when a transition containing such *op'* is received from the server. Obviously, this new version will also be generated at the server and, hence, in its due time, sent to the local site where will be used to substitute the local one.

5.8 Algorithm correctness

The consistency of a document is achieved by keeping the properties of *convergence, causality preservation* and *intention preservation.*

[5] Recall that $<n_g, v_g>$ was either $<n,v'>$ itself or one of its ancestors.

Within the algorithm presented in this article, intention preservation follows from the considerations made along its description (see section 5.6). As a consequence of this property, any document modification will be carried out by the server into the same version which the local site intended.

The achievement of convergence is clear since the server broadcasts the same transitions to all the sites. Hence, when all of them have been received, all sites will have the same contents. There is, however, a minor point concerning convergence achievement: sites will have to substitute the operations they generated locally for those (corresponding to the local ones) that will receive from the server.

The global serialization attained by the server guarantees that no causality will be violated (i.e. every pair of dependent operations will be executed in the same order in all the sites).

5.9 Document Coherence

The coherence of a document is achieved when the document contains just one version of every node. In order to achieve coherence, a merging process between all sites is required. Although this merging process usually involves some aspects that must be carried out manually, there exist a wide variety of tools and algorithms that may help in it (e.g. automatic conflict detection, several automatic or semi-automatic merging policies specified by users, assistance to conflict resolution...). [8] contains a good overview of these issues.

6 An example

This section presents an example of the behaviour of the algorithm described in the previous one. We edit the document shown in figure 4. The node at the highest level (n_1) corresponds to the document level, the node at the second level (n_2) represent a paragraph and, finally, n_4 and n_5 correspond to words. This particular document contains only one paragraph, with two words. A document structure can be defined in XML by means of DTDs and XML schemas. In this case a very simple structure has been chosen. In general, a document could be composed of chapters, sections, subsections...

Table 1 shows the requests that the server receives from different sites (in this case, sites u_1 and u_2), starting from the initial state S_0. Each request (r) is transformed into a transition (t) that will be applied to the last global state hold by the server and broadcast to all sites. The application of a transition to a global state S_i generates a new global state S_{i+1}.

The document depicted in figure 4 and the requests shown in table 1 are based on the example introduced in section 3. Figures 5 and 6 depict the document at states S_4 and S_6 respectively.

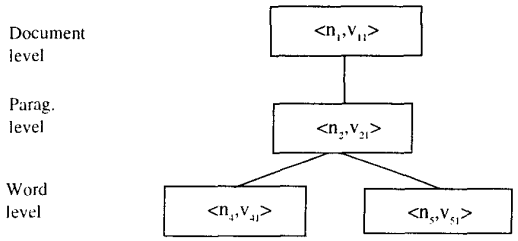

Figure 4: The document being edited at state S_0

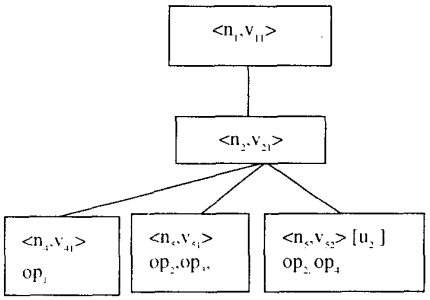

Figure 5. Document at state S_4

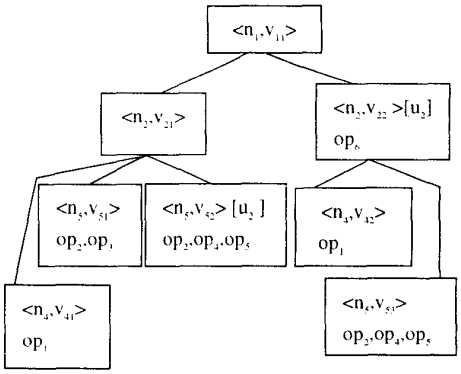

Figure 6. Document at state S_6

Table 1. Example of development

Requests - transitions	Explanation
S_0: $r=[op_1, u_1, S_0, <n_4.v_{41}>]$ $t=[op_1, u_1, <n_4.v_{41}>]$ ↓	r=Update of the word n (version v) by site u . S being the last global state seen by u at the instant of generating op_1. t=Update of the word <n .v > by u . S is obtained applying t to S_0 All r and t along the example are to be interpreted in the same way.
S_1: $r=[op_2, u_2, S_0, <n_4.v_{41}>]$ $t=[op_2, u_2, <n_4.v_{41}>]$ ↓	r=Update of $<n_4.v_{41}>$ by u_2. No conflict between $<n_4.v_{41}>$ and $<n_4.v_{41}>$.
S_2: $r=[op_3, u_1, S_2, <n_4.v_{41}>]$ $t=[op_3, u_1, <n_4.v_{41}>]$ ↓	r=Update of $<n_4.v_{41}>$ by u_1 at state S_2 No conflict since u has seen the updates that u_2 made on $<n_4.v_{41}>$
S_3: $r=[op_4, u_2, S_0, <n_4.v_{41}>]$ $t=[op_4, u_2, <n_4.v_{42}>]$ ↓	r=Update of <n .v > generated by u with last global state received $=S_0$ *Step 2*: r causes a conflict with op since u had not seen the effect of op on <n .v > at the instant of generating op_4. *Step 3*: A new version of n is created with the value of <n .v > at S : <n .v > (see fig. 5: the document at state S_3). *Step 5*: t is generated s.t. op is applied to this new version.
S_4: $r=[op_5, u_2, S_0, <n_4.v_{41}>]$ $r'=[op . u .S .<n .v >]$ $t=[op_5, u_2, <n_4.v_{42}>]$ ↓	r=Update of $<n_4.v_{41}>$ generated by u_2 with last global state received = S_0 *Step 1*: Since a new version of n has been created (from <n .v >) as a result of the processing of a u request, and since that creation is posterior to S_0, op_5 will be addressed to this new version (<n .v >). Hence the request r' will be generated. *Steps 2, 3, 4, 5*: No conflict is detected. t is generated straightforwardly.
S_5: $r=[op_6, u_2, S_2, <n_2.v_{21}>]$ $t=[op_6, u_2, <n_2.v_{22}>]$ ↓ S_6	r=Update of <n .v > generated by u with last global state received = S_2. n_2 is a paragraph. op_6 could be (for instance) the removal of the paragraph. *Step 1*: No other version of n has been created by u_2. Nothing to do. *Step 2*: r causes a conflict with op since, at the instant of generating op . u had not seen the effect of op_3 on $<n_4.v_{41}>$. *Step 3*: A new version of n is created with the value of $<n_2.v_{21}>$ at S_2: $<n_2.v_{22}>$ (see fig. 6: the document at state S_6) *Step 4*: We have to apply to the subdocument rooted at <n .v > the operations posteriors to S that u has performed on any child of <n .v >: op and op (see figure 6). op is not applied since it was performed at S and, therefore it is already included in $<n_2.v_{22}>$. *Step 5*: t is the transition generated and broadcast.

7 Conclusions and future work

We have identified some drawbacks in current algorithms for cooperative edition of text documents in real-time (which are based on operation transformations): in our opinion, they do not really achieve intention preservation and, on the other hand, transformations must be performed on operations although they may have been performed on independent document points.

We have proposed a new algorithm that, in our opinion, provides two improvements on the current ones:

- It achieves a better degree of intention preservation by keeping different versions of the updated nodes. The multi-version approach of the algorithm also makes it natural for users to decide on which version to work (if several versions of a document fragment coexist).
- It considers a hierarchical document structure which makes it possible to reduce the concurrency *conflict unit* (a collision will only occur between dependent nodes) and, at the same time, to make it easier and quicker the statement of operations on non-atomic nodes.

We are not aware of any other algorithm in the framework of cooperative edition with both features.

Our algorithm uses a server to centralize sites' requests. We have adopted this policy since it is the most natural one in the context in which we want to use it.

We are currently working on the implementation of the algorithm. We expect that the number of conflicts during a cooperative editing session will keep relatively small due to the fact that only temporally independent operations *performed on dependent nodes* cause conflicts (i.e. Several users inserting concurrently new words at different places will not collide). In contrast, in the usual transformation-oriented approaches any pair of independent operations caused a conflict and required a set of transformations.

If the number of conflicts is kept under control, the response time will be low. The overhead introduced by the client-server architecture is negligible (just two additional messages passing through the network for each operation performed at one site) compared to a peer-to-peer approach.

In the context of PROMENADE (in which we will apply the algorithm) the choice of a client-server architecture is the most natural option. However, this kind of architecture raises the fault-tolerance problem. We are working on an algorithm which is able to choose a participant to act as a server in the event of a server breakdown.

References

1. Ellis, C.A., and Gibbs, S.J, "Concurrency Control in Groupware Systems," Proc. ACM SIGMOD Conference on Management of Data, June 1989.
2. Ellis, L. and Gibbs, S. J. and Rein, G. L., "Groupware: Some Issues and Experiences", Communications of the ACM, Vol. 34, Iss. 1, pages 38-58, January 1991.
3. Extensible Markup Language (XML) 1.0. http://www.w3.org/xml
4. Greenberg, S. and Marwood, D. (1994) "Real time groupware as a distributed system: Concurrency control and its effect on the interface." in Proceedings of the ACM CSCW'94 Conference on Computer Supported Cooperative Work, p207--217, Chapel Hill, North Carolina, October 22--26.
5. Ionescu, M.; Dorohonceanu, B.; Marsic I.:A Novel Concurrency Control Algorithm in Distributed Groupware (2000) in Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000) pp. 1551-1557. Las Vegas, NV, June 2000.
6. Knister, M.; and Prakash., A.: Issues in the design of a toolkit for supporting multiple group editors. Computing Systems -- The Journal of the Usenix Association, 6(2):135--166, Spring 1993.
7. Moran, T.P.; McCall, K; et alt.: Some design principles of sharing in Tivoli, a whiteboard meeting support tool. In S. Greenberg, S. Hayne, and R. Rada, editors, Groupware for Real-time Drawing: A Designer 's guide, pages 24--36. McGraw-Hill, 1995.
8. Munson, J. P.; Dewan, P.: A Flexible Object Merging Framework. In Proc. of CSCW'94, pp. 231--241, October, Chapel Hill, NC, ACM Press (1994).
9. Nichols, D. A.; Curtis, P. et alt. High-latency, low-bandwidth windowing in the jupiter collaboration system. In Proceedings of UIST '95, pages 111--120. ACM, 1995.
10. Ressel, M.; Nitsche-Ruhland D.; Gunzenhauser, R.: "An integrating, transformation-oriented approach to concurrency control and undo in group editors," In Proc. of ACM Conference on Computer Supported Cooperative Work, pp 288-297, Nov. 1996.
11. Sun, C.; Chen, D. A Multi-version Approach to Conflict Resolution in Distributed Groupware Systems in Proc. of the 20th IEEE International COnference on Distributed Computing Systems, pp. 316-325, April 10-14, 2000.
12. Sun, C.; Ellis, C. A.: "Operational transformation in real-time group editors: issues, algorithms, and achievements, " In Proc. of ACM Conference on ComputerSupported Cooperative Work, pp.59-68, Seattle, USA, Nov. 1998.
13. Sun, C.; Jia, X. et alt.: "Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems," ACM Transactions on Computer-human Interaction, 5(1), March 1998, pp.63-108.
14. Sun, C.; Maheshwari P : An efficient distributed single-phase protocol for total and causal ordering of group operations in Proceedings of 3rd International Conference on High Performance Computing, Trivandrum India, 19-22 Dec. 1996, IEEE Computer Society, 1996, pp295-300.
15. Ribó J.M; Franch X.: PROMENADE, a PML intended to enhance standardization, expressiveness and modularity in SPM. Research Report LSI-00-34-R. Dept. LSI, Politechnical University of Catalonia (2000).
16. Ribó, J.M.; Franch, X. Building Expressive and Flexible Process Models using a UML-based Approach. In Proceedings of the 8th European Workshop in Software Process Technology. Witten, 2001 (to appear).