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 of operations such that op₂ depends on op₁, (i.e. op₂ was performed having seen the result of op₁), then op₂ is executed before op₁ 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-mentioned 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, 15] but either they did not succeed in achieving all the three correctness properties or they presented some concurrency problems (see [13] for an overview).

[11] 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
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 ‘pace’); 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: ‘pace’ 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 \( op_1 \) and \( op_2 \) 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 \( obj_1 \) 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.
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 \(<\text{node-number}, \text{version-number}>\). 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 \(<\text{node.version}>\):
- The site that has generated the conflicting operation which has resulted in the creation of the new version and whether that version is deleted 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:

- \(\text{insertCharacter}(\text{cn}, v, p, c)\) and \(\text{deleteCharacter}(\text{cn}, v, p)\). \(\text{insertCharacter}\) inserts the character \(c\) in the position \(p\) of a pair \(\text{cn}, v\) that represents a word, while \(\text{deleteCharacter}\) removes the \(p^\text{th}\) character of the word identified by \(\text{cn}, v\). Except for the pair parameter, these are the usual operations offered by many cooperative algorithms \((5, 10, 13...))\).

- \(\text{insertNode}(x, loc)\) and \(\text{removeNode}(\text{cn}, v)\). 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, \(\text{insertNode}\) 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 \(\text{removeNode}\) removes the pair identified by \(\text{cn}, v\) 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 \(u_i\) modifies a word of the document, another site \(u_j\) 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 $w_1 = \text{"abce"}$ and $w_2 = \text{"efgh"}$. See figure 1(a). Sites $i_1$ and $i_2$ begin to modify the document:

- First of all, they modify $w_1$ and $w_2$, respectively ($i_1$ inserts 'I' to $w_1$; $i_2$ inserts '2' to $w_2$). No conflict occurs (concurrent modification of independent nodes. See figure 1(b)).

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 $S_i$ received by $k$ (from the server) only in those operations performed by $k$ that the server has not included in $S_i$ (because it has not processed them yet or has processed them in a state posterior to $S_i$).

**Temporal dependency of operations.** An operation $op_i$ (performed by site $i$ at state $S_i$) is temporally dependent on an operation $op_j$ (performed by site $j$ at state $S_j$) if either:

- $i = j$ and $op_i$ has been performed after $op_j$, or
- $i = j$ and when $i$ performs $op_i$ it has already received the effect of $op_j$.

If neither $op_i$ is temporally dependent on $op_j$ nor $op_j$ is temporally dependent on $op_i$, we say that $op_i$ and $op_j$ 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 \(<n',v>\) if \(<n,v>\) belongs to the tree rooted at \(<n',v>\) (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 \(S_j\) 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 \(op\), made by a site \(k\) on a pair \(<n,v>\) may lead to a conflict in the case that some other operation \(op_i\), which is independent on \(op\), has been already performed on a pair \(<n,v>\) with some dependence with \(<n,v>\).

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 \(n_i\) will be created with the same contents seen by \(k\) immediately before \(op\) was performed. The operation \(op\) 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 \(S_j\) to the state \(S_{j+1}\) is a tuple \([op,<n,v>,u]\), where:
- \(op\) is the operation to be performed to pass from \(S_j\) to \(S_{j+1}\).
- \(<n,v>\) 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
\(x=[S_0,S_1,...,S_n]\) held by the server.

A transition \(t=[op,<n,v>,u]\) performed at state \(S_j\) is conflicting with \(req\) if:
(a) \(u\neq u\).
(b) \(i\neq j\) and \(i\leq j < k\).
(c) \(<n,v>\) is dependent on \(<n,v>\) or \(<n,v>\) is dependent on \(<n,v>\).

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

Given a request \(req=[op,u,<n,v>,S_i]\) and a sequence of global states
\(x=[S_0,S_1,...,S_n]\), 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).

\footnote{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 arising 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).

\[\begin{array}{c}
\text{\(<n,v>\)} \\
\downarrow \\
\text{\(<n,v'>\)} \\
(\text{\(u,S_i\)}) \\
\downarrow \\
\text{\(<n,v>\)} \\
\end{array}\]

*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 \(req=[op, u, S_i, <n,v>]\) reaches the server (which keeps a sequence of global states \(s=(S_i, S_{i+1}, \ldots, S_j)\), the objective is to transform that request applied locally at state \(S_i\) into a transition that will be applied to the state \(S_j\), and will generate \(S_{j+1}\). 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 \(req=[op, u, S_i, <n,v>]\), \((i \leq j)\) has created a new version \(v'\) of \(n\) (i.e. \(v'\) will be created if a conflict comes up when processing \(req\), in order to keep intention preservation).

If such a version \(v'\) has been created when processing \(req\), it becomes the version of \(n\) corresponding to \(<n,v>\) for subsequent \(u\)'s requests.
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 \( v' \):
\[
req' = [op, u, S, <n, v'>].
\]

\( S \) is the global state in which \( <n,v> \) was created (\(<r<k\>). \( S = S_1 \) 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 \( S_1, S_2, S_3 \) do not contain any conflicting transition with it. Recall that a conflicting transition with \( req' \) will be a transition \( t = [op, n, v', u] \) generated at some state \( S_j (r \leq j < k) \) by a site \( u \neq u \) on a node \( n \) such that \( n \) 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 \( <n,v'> \) has to be created (namely, \( n' \)). The new version of \( n' \) will have the value that it had at state \( S_j \).

We will call \( t \) the first conflicting transition with \( req' \) encountered within \( S_1, S_2, S_3 \) to the origin state of such a transition and \( <n,v',u> \) to the pair containing the node \( n \) a new version of which will be created.

If such conflicting transition does not exist, \( req' \) can be applied straight to \( S_1 \) in order to generate \( S_{n,v} \).

3. If the tuple \((t, S_1, <n,v>, <n,v'>)\) has been found at step 2, create a new version of \(<n,v'>\): \(<n,v'''>\)

\(<n,v'''>\) will be created with the value of \(<n,v>,<n,v'>\) at state \( S_1 \). Section 5.4 describes how to proceed to create a new version of a node at a given state.

4. If a new version \(<n,v'''>\) has been created at step 3, apply to it all the u’s remaining operations

Subsequent states to \( S \) may contain transitions performed by \( u \) on some other pair of which \(<n,v'>\) 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 \(<n,v'>\) before including \( op \).

Hence, in this step, if a new version \(<n,v'''>\) has been created in step 3, we apply to it all the operations coming from \( u \), that have been performed at states posterior to \( S \) in any pair \(<n,w>\) such that \(<n,v'>\) is an ancestor of \(<n,w>\).

5. Apply \( op \) to the right version of \( n \) at the state \( S \), and do the broadcast

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

Otherwise \( op \) will be applied to the corresponding version of \(<n,v'>\) within \(<n,v'''>\) (namely \(<n,v'''>\) itself or one of its ancestors).

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 it 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.

\[\text{Recall that } <n,v'> \text{ was either } <n,v> \text{ itself or one of its ancestors.}\]
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 \( n_2 \) represents a paragraph and, finally, \( n_3 \) and \( n_4 \) 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 \( s_i \) and \( s_j \)), 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 \( S_g \) generates a new global state \( S_{g+1} \).

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 \( S_0 \) and \( S_n \), respectively.

Figure 4: The document being edited at state \( S_0 \)

![Diagram for state \( S_0 \)](image)

Figure 5: Document at state \( S_1 \)

![Diagram for state \( S_1 \)](image)

Figure 6: Document at state \( S_n \)

![Diagram for state \( S_n \)](image)
Table 1. Example of development

<table>
<thead>
<tr>
<th>Requests - transitions</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r = \langle \text{op}, u, S, \langle n, v \rangle \rangle )</td>
<td>( \Rightarrow ) Update of the word ( n ) (version ( v )) by site ( u ) ( \cdot S ) being the last global state seen by ( u ) at the instant of generating ( \text{op} ).</td>
</tr>
<tr>
<td>( t = \langle \text{op}, u, \langle n, v \rangle \rangle )</td>
<td>( \Rightarrow ) Update of the word ( \langle n, v \rangle ) by ( u ) ( \cdot S ) is obtained applying ( t ) to ( S ).</td>
</tr>
<tr>
<td>( r = \langle \text{op}, u, S, \langle n, v \rangle \rangle )</td>
<td>( \Rightarrow ) Update of ( \langle n, v \rangle ) by ( u ).</td>
</tr>
<tr>
<td>( t = \langle \text{op}, u, \langle n, v \rangle \rangle )</td>
<td>( \Rightarrow ) Update of ( \langle n, v \rangle ) by ( u ) at state ( S ).</td>
</tr>
</tbody>
</table>

\( S \); \( r = \langle \text{op}, u, S, \langle n, v \rangle \rangle \)

No conflict between \( \langle n, v \rangle \) and \( \langle n, v \rangle \).

\( S \); \( r = \langle \text{op}, u, S, \langle n, v \rangle \rangle \)

No conflict since \( u \) has seen the updates that \( u \) made on \( \langle n, v \rangle \).

\( S \); \( r = \langle \text{op}, u, S, \langle n, v \rangle \rangle \)

\( \Rightarrow \) Update of \( \langle n, v \rangle \) generated by \( u \) with last global state received \( S \).

\( t = \langle \text{op}, u, \langle n, v \rangle \rangle \)

\( \Rightarrow \) Fragment \( S \), \( \langle n, v \rangle \rangle \) generated by \( u \) with last global state received \( S \). Hence the request \( t \) will be generated.

\( S \); \( r = \langle \text{op}, u, S, \langle n, v \rangle \rangle \)

\( \Rightarrow \) Update of \( \langle n, v \rangle \) generated by \( u \) at state \( S \).

\( t = \langle \text{op}, u, \langle n, v \rangle \rangle \)

\( \Rightarrow \) Fragment \( S \), \( \langle n, v \rangle \rangle \) generated by \( u \) with last global state received \( S \).

\( S \); \( r = \langle \text{op}, u, S, \langle n, v \rangle \rangle \)

\( \Rightarrow \) Update of \( \langle n, v \rangle \) generated by \( u \) at state \( S \).

\( t = \langle \text{op}, u, \langle n, v \rangle \rangle \)

\( \Rightarrow \) Fragment \( S \), \( \langle n, v \rangle \rangle \) generated by \( u \) with last global state received \( S \).

\( S \); \( r = \langle \text{op}, u, S, \langle n, v \rangle \rangle \)

\( \Rightarrow \) Update of \( \langle n, v \rangle \) generated by \( u \) at state \( S \).

\( t = \langle \text{op}, u, \langle n, v \rangle \rangle \)

\( \Rightarrow \) Fragment \( S \), \( \langle n, v \rangle \rangle \) generated by \( u \) with last global state received \( S \).

<table>
<thead>
<tr>
<th>Requests - transitions</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r = \langle \text{op}, u, S, \langle n, v \rangle \rangle )</td>
<td>( \Rightarrow ) Update of the word ( n ) (version ( v )) by site ( u ) ( \cdot S ) being the last global state seen by ( u ) at the instant of generating ( \text{op} ).</td>
</tr>
<tr>
<td>( t = \langle \text{op}, u, \langle n, v \rangle \rangle )</td>
<td>( \Rightarrow ) Update of the word ( \langle n, v \rangle ) by ( u ) ( \cdot S ) is obtained applying ( t ) to ( S ).</td>
</tr>
<tr>
<td>All ( r ) and ( t ) along the example are to be interpreted in the same way.</td>
<td></td>
</tr>
</tbody>
</table>

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).

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

3. Extensible Markup Language (XML) 1.0. http://www.w3.org/xml