International Journal of Computer Science and Applications
©Technomathematics Research Foundation
Vol. 9, No. 3, pp. 47 – 60, 2012

SIMPLIFIED CENTRALIZED OPERATIONAL TRANSFORMATION ALGORITHM FOR CONCURRENT COLLABORATIVE SYSTEMS

ANDRZEJ ROMANOWSKI

Computer Engineering Department, Technical University of Lodz
Stefanowskiego 18/22, 90-924 Lodz, Poland
androm@kis.p.lodz.pl

PAWEL WOZNIAK

Computer Engineering Department, Technical University of Lodz
Stefanowskiego 18/22, 90-924 Lodz, Poland
pawel@ubicomp.pl

JULIUSZ GONERA

Computer Engineering Department, Technical University of Lodz
Stefanowskiego 18/22, 90-924 Lodz, Poland

This paper describes an algorithm devised for collaborative text editor. Each user of the editor can see the changes made by others in real-time and make their own changes without any delay (i.e. there is no delay between typing and the text appearing on the screen).

The algorithm described in this paper belongs to the group of operational transform (OT) algorithms and it expands the ideas from some existing solutions [2, 5, 6]. Unlike most of other algorithms, it has been devised for centralized systems (i.e. requires a central server to coordinate the transformations). This, together with the undo/do/redo scheme [5, 6] removes the requirement of satisfying the so-called Transformation Properties, thus reducing the complexity of the transformation function.

The proposed algorithm was developed for plain text documents, but it can be extended to accommodate more operations or even used with data other than text. Even though it was developed for a web application, it can be also used in desktop solutions with a central server.

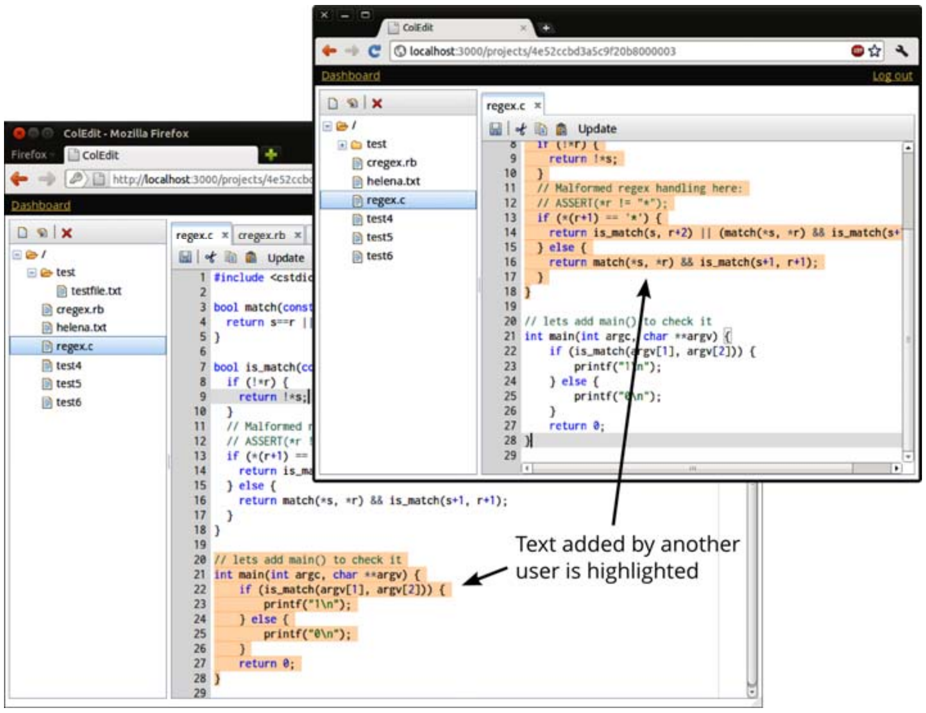
Keywords: Operational transform, groupware, collaborative workspace

1. Collaborative Systems

Computer systems used to facilitate collaboration between many people, such as collaborative text editors, are called groupware systems.

Those systems, where the actions of one user must be immediately visible by the other users, are characterized by two parameters, *response time* and *notification time*. Response time is the time necessary for the actions of a user to be reflected in their own interface, while notification time is the time necessary for those actions to be propagated to the other users [1]. In collaborative text editors, the response time must be as close to zero as possible. It is unacceptable for the user of such an editor to wait until the text they type

appears on the screen. The notification time is dependent on the network delay and usually cannot be minimized.
The main purpose of collaborative systems is facilitating their users to work on a common goal or project. Such systems allow the participants not only to communicate, like messaging or VoIP systems do, but also provide a common workspace that all the users can interact with in order to create a document of some kind (text, image, audio, etc.). One of the first collaborative applications was GRoup Outline Viewing Editor (GROVE) [1], developed in 1989, which was a simple text editor, which allowed several people to write notes in the form of lists.


ColEdit - Mozilla Firefox
Firefox
ColEdit
http://localhost:3000/projects/4e52ccbc
Dashboard
/
test
testfile.txt
cregex.rb
helena.txt
regex.c
test4
test5
test6
regex.c
cregex.rb
Update
ColEdit
localhost:3000/projects/4e52ccbd3a5c9f20b8000003
Dashboard
Log out
Text added by another user is highlighted

```
#include <cstdi

bool match(const
  return s==r |
}

bool is_match(c
  if (!*r) {
    return !*s;
  }
  // Malformed r
  // ASSERT(*r
  if (*(r+1) ==
    return is_ma
  } else {
    return match(*s, *r) && is_match(s+1, r+1);
  }
}

// lets add main() to check it
int main(int argc, char **argv) {
    if (is_match(argv[1], argv[2])) {
        printf("1\n");
    } else {
        printf("0\n");
    }
    return 0;
}
```

```
  if (!*r) {
    return !*s;
  }
  // Malformed regex handling here:
  // ASSERT(*r != "*");
  if (*(r+1) == '*') {
    return is_match(s, r+2) || (match(*s, *r) && is_match(s+
  } else {
    return match(*s, *r) && is_match(s+1, r+1);
  }
}

// lets add main() to check it
int main(int argc, char **argv) {
    if (is_match(argv[1], argv[2])) {
        printf("1\n");
    } else {
        printf("0\n");
    }
    return 0;
}
```

Figure 1. The web application for which the algorithm was devised (two windows representing two different user sessions with the same document)

Collaborative applications started to gain attention with the development of Internet and web technologies. One of the most important developments in this field was Etherpad, which allowed many people to edit a common text document. Because of its popularity it was acquired by Google and its creators implemented the collaborative features of Google Wave and Google Docs[1]. The aforementioned web-based solutions allow the

[1] http://techcrunch.com/2009/12/04/google-acquires-etherpad/

users to collaborate on typical text data by providing such tools as lists, tables or text formatting, but are not suitable for editing source code. The algorithm proposed in this paper has been devised for a web-based source code editor (see Fig. 1), which could be useful for programmers thanks to providing such features as syntax highlighting and saving the files on the user's hard disk instead of storing them only on the server. Such an editor could be useful in agile pair programming, phone interviews with live coding, distance learning programming courses or live editing documents that have many authors (e.g. LaTeX publications).

2. Concurrency Issues

When a system is concurrent, many users can make changes to it at the same time. Problems arise when those changes are conflicting, which is often the case when editing text collaboratively.

One can imagine that two users, Alice and Bob, have an empty document opened in a collaborative editor. Alice types "A" (i.e. insert character "A" at offset 0), while Bob at (more or less) the same time types "B" (i.e. insert character "B" at offset 0). Because of nondeterministic communication latency they do not see each other's changes immediately. Therefore, they both see an empty document when they start typing. After typing, Alice has a document with content "A" and Bob has a document with content "B". When Alice receives Bob's operation, her document will be "BA" (Bob inserted "B" at offset 0). When Bob receives Alice's operation, his document will be "AB" (Alice inserted "A" at offset 0). This is just the simplest example of conflicting user actions, but even in such case, the documents of two users diverge – illustration is shown on Fig. 2.

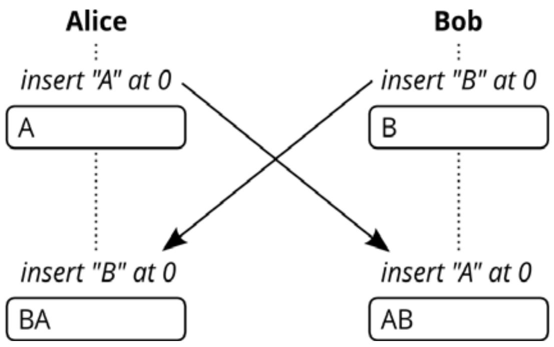


Figure 2. Conflicting operations cause the documents to diverge

These conflicts have to be resolved in some way. The simplest solution would be to lock the document and let only one user "have the floor," while other users could only observe.

Another obvious solution would be to use a local locking mechanism. Since the editor which uses the algorithm described in this paper is a source code editor, it could for example lock the whole line of code when a user moves its cursor to it, so that only one user can edit this line. This solution is better, but still has some important disadvantages. First of all, there might be cases when two users would like to edit the same line. In regular programming languages this should not happen often, but in mark-up languages, such as HTML or LaTeX, a single line can contain a whole paragraph of text. Moreover, the user can go away from keyboard and then nobody could edit the line until they are back or until some kind of timeout is reached. Finally, an ID would have to be generated for each line as line numbers change when editing the document.

3. Operational Transformation

Because of the drawbacks of the aforementioned solutions, a new technique used for consistency maintenance has been invented, called Operational Transformation. The main principle of OT is that all the actions of the user are executed immediately in their own user interface and appropriately *transformed* in other users' interfaces, so that the documents converge to the same state.

OT was pioneered by the GROVE system in 1989 [1]. It is most often utilised for concurrency control in *real-time groupware systems*. A real-time groupware system is a multi-user system where the actions of one user have to be quickly propagated to the other users. In particular, a groupware editor allows its users to view and edit the same document (e.g. text or graphics) collaboratively, at the same time from multiple sites connected by communication networks.

In OT user actions result in operations which are executed on the document. A single operation can be for example inserting a letter in a text document. Operations by different users working on the same part of the document are transformed to avoid conflicts. Importantly, OT algorithms are optimistic, i.e. they achieve synchronization without using locks and delaying the execution of local user actions. This is crucial in editors as it would be unacceptable for users to. wait for the text to appear on the screen after they typed it.

3.1. *Review of Existing Algorithms*

Consistency maintenance is a significant challenge when designing and implementing an OT algorithm for real-time collaborative systems. For the past two decades, after the original algorithm, called dOPT [1], have been proven not to be fully correct, many different approaches have been devised by different authors to solve this problem. While the vast majority of those solutions aimed to develop an algorithm for fully distributed peer-to-peer systems [1, 3, 5, 6], fewer described algorithms for centralized systems that use a central server as a coordinator [2].

The dOPT algorithm fails in the case of the so-called dOPT puzzle, when an operation is concurrent with two or more other operations from different sites [4]. There are two widely-known alternative distributed algorithms which fix dOPT's problem, adOPTed [3] and GOT/REDUCE [5, 6].

dOPT requires the transformation function T to satisfy a property called Transformation Property 1 (TP1):

$$o_a \circ o_b' \equiv o_b \circ o_a' \tag{1}$$

where $o_a' = T(o_a, o_b)$ (operation o_a transformed by o_b) and $o_b' = T(o_b, o_a)$ (vice-versa). In other words, the effect of first applying operation o_a and then operation o_b must be the same as the effect of first applying o_b and then o_a. adOPTed solution consists of adding one more requirement to the transformation function, called Transformation Property 2 (TP2), which states that for any operation o:

$$T(T(o, o_a), o_b') = T(T(o, o_b), o_a') \tag{2}$$

Additionally, adOPTed stores the operations history in a N-dimensional interaction model graph instead of a linear log, where N is the number of cooperating sites. Even though those changes solve the problems encountered in dOPT, they make the adOPTed algorithm significantly more complex.

Another distributed approach that improves dOPT is GOT, developed as a part of the REDUCE system. Thanks to employing the undo/do/redo scheme, this algorithm requires neither TP1 nor TP2 to be satisfied by the transformation function. Nonetheless, for this scheme to work in a distributed environment, another type of transformation, called Exclusion Transformation (ET) had to be introduced apart from the traditional Inclusion Transformation (IT). ET transforms an operation o_a that already depends on another operation o_b in such way that the impact of o_b is effectively excluded from o_a. In practice, this again introduces additional complexity since two transformation functions are required instead of a signle one.

Finally, the OT algorithm of the Jupiter Collaboration System [2] takes a notably different approach to solving issues encountered in dOPT. Instead of being a fully distributed system, it uses a central server to coordinate the communication between users. Even though such an architecture can be less versatile (not for mission critical systems – if the central server fails, the communication is interrupted), it is still suitable in many cases (e.g. intranet and web applications) and, more importantly, it greatly simplifies the OT algorithm. The resulting two-way communication (clients communicate only with the server) eliminates the concern for ensuring causality among many cooperating sites, required by other algorithms (the server decides the order of operations). Therefore, although Jupiter does not use the undo/do/redo scheme, its transformation function only has to satisfy TP1 and not TP2. Still, satisfying TP1 can also result in a more complex transformation function. Furthermore, to store the history of operations Jupiter requires a two-dimensional state space graph for each client instead of a single linear log, which can be memory-intensive. An alternative approach to avoid this requirement has been devised by Google [7]. It consists of forcing the client to wait for the acknowledgement from the server before sending next operations. In this way, the documents on the server and on the client do not differ by more than one state. Further details on comparison of OT algorithms can be found in Chengzheng and Clarence [4].

3.2. *Assumptions and Basic Principles for Proposed Algorithm*

In order to minimize the complexity of the collaborative system while maintaining the benefits of OT, a new algorithm has been devised. It is based on the ideas from the algorithms mentioned in the previous section.

The proposed algorithm, called Simplified Centralized Operational Transformation (SCOT), requires a central server to coordinate the collaboration. While distributed collaboration systems prevail in current applications, the algorithm is equally useful for cloud-based computations. The abstract solution descried in this paper can be implemented without the use of a central server with one of the users or a cloud service acting at the coordinating agent. It allows collaboration between multiple users, but the communication takes place only between the client and the server (two-way communication between two parties).

To simplify the algorithm, we assume the clients send new messages (with operations) only after receiving a response acknowledging the previous message. We also assume that the server can process only one message at a time, i.e. processing it is either atomic or there exists a locking mechanism, and that the transport layer (e.g. TCP) delivers messages in order.

The algorithm operates on text documents represented as a single string and supports two operations denoted in the following way:

ins(p, t), which inserts string *t* at document position *p*,

del(p, l), which deletes *l* characters at document position *p*.

Both operations have to be represented in such a way that they contain enough information to be reverted, i.e. a *del* operation cannot store only the position and length, but it also has to store the string it deletes.

The document is stored on the server and replicated on all clients. Its state is determined by a *revision* number. The revision of a new (but not necessarily empty) document is zero and is increased by one after each *changeset* applied to the document. Changesets group operations exchanged between the server and the clients in chronological order. Each changeset can contain an arbitrary number of operations and has a *base revision* number, which determines the document state to which the changeset can be applied. A changeset with base revision *n* can be applied only to the document with revision number equal to *n*. In other words, the revision number defines the context for operations.

Operation's base revision is equal to the base revision of the containing changeset. Applying a changeset to the document means applying all its operations to the document

in order. Changesets applied by the server are stored in chronological order in a *changeset log*. Changesets contain their author's ID, which can be either sent by the client or injected by the server.
Clients store only two changesets: the *pending changeset* and the *sent changeset*. Pending operations are accumulated in the pending changeset until they can be sent. When the pending changeset is sent, it is renamed as a new sent changeset and a new empty pending changeset is created. When the client receives an acknowledgement message, the sent changeset is emptied and the document revision is increased by one.
Since the coordinating server cannot release any operations itself, all the examples in the following algorithm description will include two clients and the server.

4. The Algorithm

This section explains the principles of algorithm works. The algorithm, due to its client-server design, is divided into two distinct components. First, the algorithm that runs on the central server is described, and then the client-side part is discussed.

4.1. *Server-side Transformations*

First, we analyse a simple case where two clients (users), Alice and Bob, begin editing a document with string "FOO" (fig. 3). They both start typing at the same time, Alice generates ins(0, "A") and Bob generates ins(3, "B"). The server first receives Alice's changeset and applies it immediately because its base revision (i.e. 0) is equal to the document revision on the server. Then, Bob's changeset is received. Since its base revision (i.e. 0) is lower than the document revision on the server (i.e. 1), the changeset's operation cannot be applied. The server's document context is not the same as the context in which Bob's operation was originally generated.

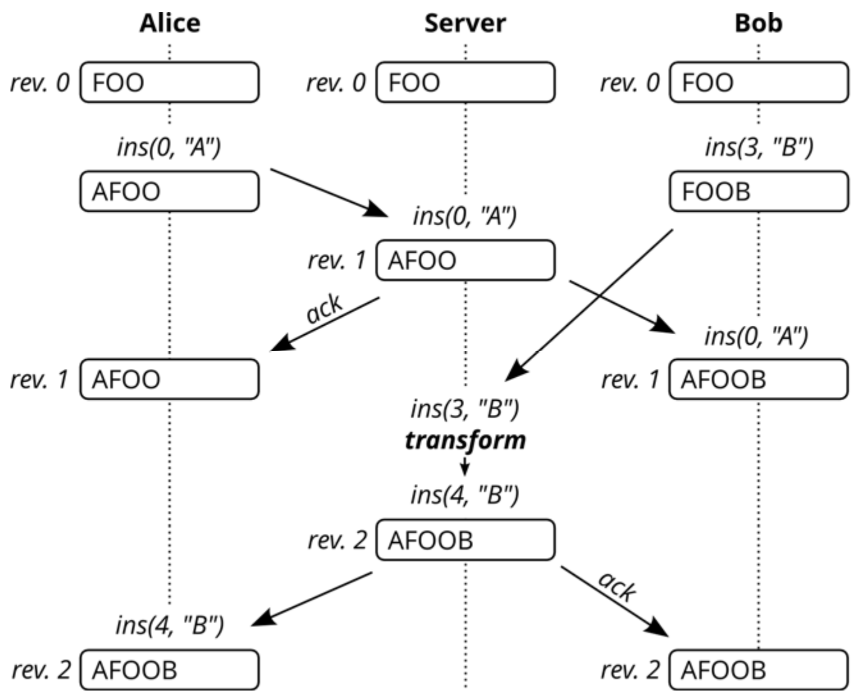


Figure 3. An example of a simple operation conflict resolved on the server

If Bob's operation ins(3, "B") were applied without transforming it to the document with content "AFOO", the resulting document would be "AFOBO", which is not what Bob meant. The intention of the operation, inserting letter "B" after the string "FOO", would not be preserved.

Bob's changeset and the contained operation have to be *transformed*. Intuitively, Alice's operation should transform Bob's operation into *ins(4, "B")* because Alice's operation moved the position where Bob wanted to insert "B" by one. The algorithm defines a special transformation function for this purpose (eq. 8.1), which will be described in more detail in section 4.

$$transformOp(o, o_t) = o' \qquad (3)$$

Given two operations o and o_t, this function transforms o using o_t and returns a new operation o'.[2]

[2] In reality, the transformation function can also return two or no operations in some specific cases. This is intentionally left out to simplify the description; it will be mentioned later.

```
**procedure** $transformCh(c, c_t)$
    $ops \leftarrow c$'s operations
    $ops_t \leftarrow c_t$'s operations
    **for all** $o \in ops_t$ **do**
        **for all** $o_t \in ops_t$ **do**
            $o \leftarrow transformOp(o, o_t)$
        **end for**
    **end for**
    increase $c$'s base revision by 1
**end procedure**
```

Operations are transformed when their containing changesets are transformed. A changeset can be transformed with another changeset only if they are independent, i.e. their base revisions are equal (which is the case in the presented situation). Transforming a changeset c with another changeset c_t means transforming in order all the operations of changeset c with the operations of changeset c_t. The base revision of changeset c is increased by one (fig. 4).

To sum up, when the server receives a changeset c it first checks if its base revision is equal to the revision of the document. If it is so, it simply applies the operations of c to the document, adds c to the changeset log, increases document revision by one and sends c to the other clients (i.e. all the clients except the author). If c's base revision is lower than the document revision, c is first transformed in order with all the changesets from the changeset log such that their base revision is greater or equal to c's base revision.

4.2. *Client-side Transformations*

The above example required operations to be transformed only on the server. Transformed operations were then forwarded to other clients and applied by them without any transformations. Very often this is not enough and the clients also have to transform operations. Let us imagine the situation presented in figure 5.

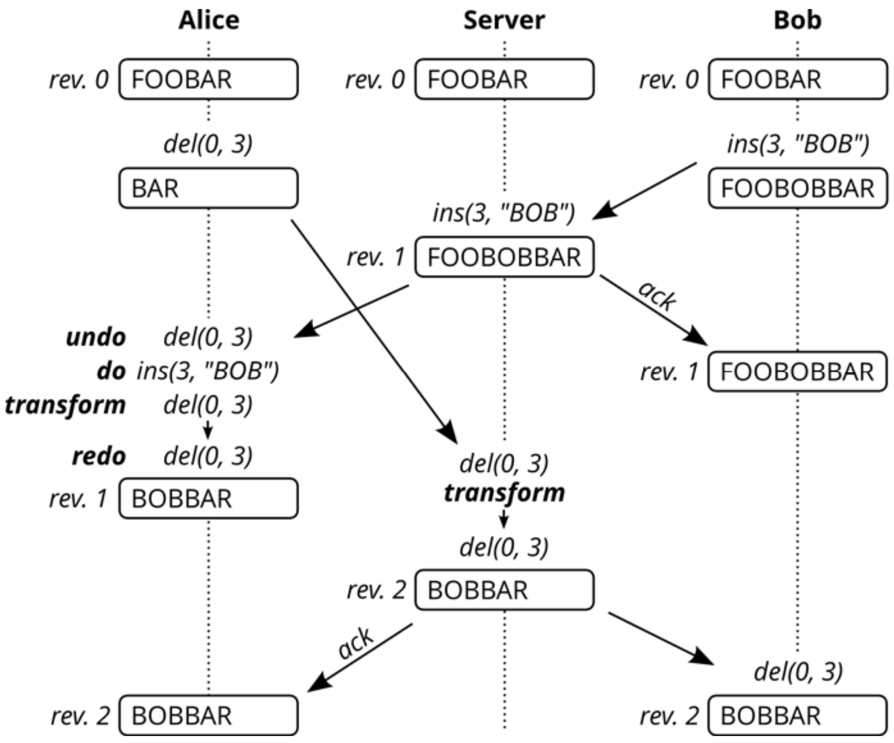


Figure 5. Operation conflict resolved on the client side

This time, Bob's changeset with operation *ins(3, "BOB")* reached the server first and is stored in server's changeset log without any transformation. This changeset is then forwarded to Alice. When Alice receives Bob's changeset, it cannot be applied directly to her document because the intention would not be preserved (it would result in a string "BARBOB", but Bob wanted to add the string "BOB" before "BAR"). To avoid this, Bob's changeset could be transformed with Alice's sent changeset and pending changeset. This approach, however, would mean that the combinations of operations transformed on the server and on the client would be different. As a result, it would impose an asymmetry between the transformation function on the server and on the client.

There exists another approach which does not introduce such complications (fig. 7). Instead, when Alice receives Bob's changeset, her pending and then sent changesets (in such reversed order) are *undone* and Bob's changeset is applied to the document in the previous state. Then, Alice's pending and sent changesets are transformed with Bob's changeset[3], and *redone* to the document (in normal order, first sent, then pending). In this

[3] In this particular example, however, this transformation does not change Alice's operation because Bob's operation has higher position than Alice's operation.

way, the transformations on Alice's document reflect the transformations on the server. This process must be implemented in such a way that it is invisible for the user.
Later, to finish the example, Alice's changeset with operation *del(0, 3)* is received by the server. Comparing the base revisions of both changesets, we conclude that they are independent (base revisions are equal). Therefore, Alice's changeset has to be transformed with Bob's changeset before it can be added to the changeset log (there can be no two changesets with the same base revision in the log). Transforming Alice's operation with Bob's operation results in the same operation because Bob's operation's position is greater than the end of the range on which Alice's operation performs. Therefore, Alice's transformed changeset will differ from the original one only by the base revision number. After propagating Alice's changeset to Bob, the three parties end up with the same, consistent document.

4.3. *The Transformation Function Details*

Previously, it was stated that the transformation function transforms a given operation *o* using another operation o_t and returns new operation *o'* (eq. 3). This was a simplification. In fact, the transformation function returns a list of operations containing one (the majority of cases), two or no operations as seen in eq. 4.

$$transformOp(o,o_t) = [(o_1',(o_2'))] \tag{4}$$

In all the cases the resulting operations are transformed with respect to the original operation *o* in such a way that they can be applied to the document state obtained by first applying the transforming operation o_t to the original document state.
To facilitate the description of the transformation function, the following additional functions are introduced:
pos(o), returning the position of the operation,
len(o), returning the length of the string on which the operation performs,
end(o), returning the end position of the operation, i.e. *pos(o) + len(o)*.
There is also one important difference between *ins* operations and *del* operations. *ins* operations should be perceived as *point* operations (they operate on some point, position of the existing document), while *del* operations should be perceived as *range* operations (they operate on some range of the existing document). This implies that *del* operations can overlap with other independent operations or even enclose other independent operations.
First of all, an operation *o* is only transformed by another operation o_t if o_t operates on the text before or inside of *o*. Therefore, the necessary condition for the operation *o* to be transformed by operation o_t is:

$$pos(o_t) < end(o) \tag{5}$$

As observed in the algorithm description (section 3), *ins* operations and *del* operations transform operations in a different way. Generally speaking, when the transforming operation o_t and the transformed operation o do not overlap, o is shifted right by $len(o_t)$ if o_t is an *ins* operation or shifted left by $len(o_t)$ if o_t is a *del* operation.
However, if the transformed and transforming operations overlap (which can occur only when at least one of them is a *del* operation), there are more critical cases. One of those cases results in two new operations being returned instead of only one operation. This happens when a *del* operation o is transformed by an *ins* operation o_t such that o_t is enclosed by o, i.e.:

$$pos(o) < pos(o_t) < end(o) \tag{6}$$

In such a case o_t splits o into two new *del* operations which do not delete the text inserted by o_t and thus o's intention is preserved.
Another case that is worth mentioning is the one when the transformation function returns an empty list (no operation). This happens when a *del* operation o is transformed by a *del* operation o_t such that o_t encloses o, i.e.:

$$pos(o) \geq pos(o_t) \quad \wedge \quad end(o) \leq end(o_t) \tag{7}$$

In such situation, the operation o_t already deletes the characters that o would delete and therefore o can be discarded.

5. Comparison to Other Solutions and Discussion

The presented SCOT algorithm is based on some ideas from the paper
"*High-latency, low-bandwidth windowing in the Jupiter collaboration system*" [2].
Like the Jupiter system, this algorithm is also a client-server solution, however, it introduces two important changes.
First of all, the SCOT algorithm forces clients to wait for an acknowledgement from the server before sending further messages, similarly to the Google Wave OT [7]. Thanks to this feature, the server does not have to store a state space graph for each connected client like Jupiter does. Instead, in the presented algorithm, the server can store a linear changeset log. Even though waiting for acknowledgements may seem like a limitation, in practice it helps to considerably reduce the complexity and memory requirements of the server algorithm. One trade-off of this solution is that clients will receive operations of other clients in small chunks in intervals of the round trip time between the client and the server. In practice, however, it should not influence the collaboration noticeably if the client-server latency is not very high.
The part of the SCOT algorithm, which is responsible for receiving changesets by clients, is the second noteworthy difference from the Jupiter system. It employs a kind of an undo/do/redo scheme used in the GOT algorithm from the REDUCE system [5, 6, 4]. In case of our algorithm this scheme is, however, significantly simpler because of the client-

server model. Thanks to the centralized architecture (clients communicate only with the server), the server can be used for total ordering of operations (operations on the client machines are always transformed in the same order as on the server). Therefore, there is no need for exclusion transformations and an undo/transform-do/transform-redo scheme. Instead, a scheme that could be described as undo/do/transform-redo has been implemented where the operations received from the server are never transformed by the client. Consequently, the client is responsible for transforming its sent and pending operations so that they obey the total ordering of operations established by the server.

Most OT algorithms depend on the specific properties of the transformation function (TP1 and TP2), which can make the transformation function more complex. It is also often hard to verify if a given transformation function satisfies those properties without examining all the possible cases of transformations. As shown, some seemingly correct transformation functions do not really satisfy TP1 and TP2 [6]. Fortunately, by using the total ordering of operations (i.e. the operations are always executed in the same order on the server and all the clients), none of those properties need to be satisfied in the SCOT algorithm for the resulting document to always converge to the same state on all the sites.

The presented algorithm's advantage over distributed solutions is its highly reduced computational complexity. Whereas distributed systems can accommodate more users and may provide extended stability, the computational costs of this approach is quite severe. Consequently, the authors recommend utilising the solution developed in this paper in cloud computing applications.

6. Conclusions and Future Development

This paper presented a new Operational Transformation algorithm, SCOT, based on the algorithm developed for the Jupiter system [2] and the GOT algorithm [5, 6]. The main reason for developing the SCOT algorithm was providing a simpler alternative to the existing OT solutions, which would match the centralized architecture of web applications. Since web applications are becoming increasingly popular, such an algorithm can have a lot of uses, not only in text editors, but also in collaborative wikis or even web-based collaborative graphics editors. Nonetheless, the described algorithm is suitable for any kind of application as long as its architecture involves a central server that can be used to coordinate the concurrency control.

Future work includes adapting the SCOT algorithm to work with data other than text (e.g. graphics). The first step to achieve this is developing a transformation function that supports all the necessary operations for the given type of data. Since the transformation function in the SCOT algorithm does not have to satisfy TP1 and TP2, developing new transformation functions should be intuitive. In theory, operating on more complex data should not pose any difficulties, but real implementation should be tested to guarantee that no other modifications to the SCOT algorithm are required in such cases.

Furthermore, even though the SCOT algorithm has been implemented and used in a prototype of a web-based collaborative text editor, more research is needed in real-life collaborative environments exposed to a wider array of users. Such a study would allow us to verify if the design of the algorithm is appropriate for real-life use cases. It would help us understand how we can improve it to suit better both the architecture of centralized systems, in particular web applications, and the expectations of the users of those systems.

7. References

(1) C.A. Ellis and S.J. Gibbs. *"Concurency Control in Groupware Systems"*. In: Proceedings of the 1989 ACM SIGMOD international conference on Management of data. SIGMOD '89. Portland, Oregon, United States, 1989, pp. 399–407.

(2) David A. Nichols et al. *"High-latency, low-bandwidth windowing in the Jupiter collaboration system"*. In: Proceedings of the 8th annual ACM symposium on User interface and software technology. UIST '95. Pittsburgh, Pennsylvania, United States, 1995, pp. 111–120.

(3) Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. *"An integrating, transformation-oriented approach to concurrency control and undo in group editors"*. In: Proceedings of the 1996 ACM conference on Computer supported cooperative work. CSCW '96. Boston, Massachusetts, United States, 1996, pp. 288–297.

(4) Chengzheng Sun and Clarence Ellis. *"Operational transformation in real-time group editors: issues, algorithms, and achievements"*. In: Proceedings of the 1998 ACM conference on Computer supported cooperative work. CSCW '98. Seattle, Washington, United States, 1998, pp. 59–68.

(5) Chengzheng Sun et al. *"A Consistency Model and Supporting Schemes for Real-time Cooperative Editing Systems"*. In: Proceedings of the 19th Australian Computer Science Conference. Melbourne, Australia, 1996, pp. 582–591.

(6) Chengzheng Sun et al. *"Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems"*. In: ACM Trans. Comput.-Hum. Interact. (1998), pp. 63–108.

(7) David Wang, Alex Mah, and Soren Lassen. *Google Wave Operational Transformation*. July 2010. URL : http://wave-protocol.googlecode.com/hg/whitepapers/operational-transform/operational-transform.html (visited on 06/14/2011).

(8) Du Li and Rui Li. An approach to ensuring consistency in peer-to-peer real-time group editors. Computer Supported Cooperative Work (CSCW), 17(5-6):553–611, 2006.

(9) Grishchenko V., Deep Hypertext with Embedded Revision Control Implemented in Regular Expressions, WikiSym '10, Gdansk, Poland, 2010