THE FIRST CORRECT CONCURRENCY CONTROL ALGORITHM

JAMES SMITH

e-mail address: jecs@imperial.ac.uk

Imperial College London

Abstract. We present for the first time a complete solution to the problem of proving the correctness of a concurrency control algorithm for collaborative text editors against the standard consistency model. The success of our approach stems from the use of comprehensive stringwise operational transformations, which appear to have escaped a formal treatment until now. Because these transformations sometimes lead to an increase in the number of operations as they are transformed, we cannot use inductive methods and adopt the novel idea of decreasing diagrams instead. We also base our algorithm on a client-server model rather than a peer-to-peer one, which leads to the correct application of operational transformations to both newly generated and pending operations. And lastly we solve the problem of latency, so that our algorithm works perfectly in practice. The result of these innovations is the first ever formally correct concurrency control algorithm for collaborative text editors together with a fast, fault tolerant and highly scalable implementation.

1. Introduction

Collaborative text editors have something of a convoluted history. The idea was first publicly mooted in 1968 by Turing Award winner Douglas Engelbart in his landmark demo that posthumously became known as "The Mother of all Demos" [Eng68]. Some twenty years later the first paper on a concurrency control algorithm appeared, although no correctness proof was given [EG89]. Indeed the algorithm was found to be incorrect and partial alternatives were proposed, this time along with correctness proofs [Cor95, RNRG96]. The standard consistency model was also defined around this time [SYZC96]. However, in spite of the plethora of algorithms and implementations that followed, the problem of proving the correctness of a concurrency control algorithm against this consistency model seems never to have been solved.

We briefly outline some of the issues behind this. To begin with, formally correct characterwise operational transformations remained elusive until relatively recently [IOR03] whilst formally correct stringwise operational transformations cannot be found in the literature at all. We think the unreasonable correctness criteria put upon operational transformations by peer-to-peer algorithms in particular are part of the reason for this. In

2012 ACM CCS: [**Theory of computation**]: Design and analysis of algorithms; Concurrent algorithms / [**Information systems**]: Data management systems; Information systems applications—Collaborative and social computing systems and tools—Synchronous editors.

Key words and phrases: collaborative text editor, operational transformation, concurrency control algorithm, consistency model.

fact despite their inherent complexity peer-to-peer algorithms have nearly always been preferred [VCFS00, OUI05, WUM10] and have persisted up until the present. Other consistency models have now also been proposed [LL05, LL07, SS09, LPS09, LL10], no doubt in response to an inability to prove the correctness of any algorithm against the standard one, and these cloud the picture. Lastly, modern collaborative platforms such as Google Docs require complex data types other than plain text, and this complicates things still further.

Our solution addresses all of these issues. Firstly, we give a common-sense and comprehensive definition of stringwise operational transformations, we think for the first time. Secondly, we base our algorithm on the client-server model, which we feel is more appropriate to a modern Internet setting. Thirdly, when we prove correctness against a consistency model, we do so against the standard one, which, if a consistency model is needed at all, is adequate. Lastly, we work only with plain text documents and their attendant inserts and deletes, but solve this problem completely.

In what follows, for the most part we give the details of our algorithm and demonstrate its correctness *first*, before outlining the concepts and contributions to be found elsewhere. Our reason is this: our algorithm was conceived in a vacuum, so to speak, without knowledge of the surrounding literature, and we think that this approach contributed at least in part to a successful outcome. Whilst we do not espouse such an approach in general, we nonetheless feel that it has it merits, and we feel that for this reason it is more natural to present our algorithm in line with the way in which it was conceived. We also hope that its correctness can be shown to be self-evident without recourse to consistency models and the like.

Finally a note on the naming of our algorithm and its utility. We chose 'Concur' because as well as being a fragment of 'concurrency', it is also an antonym of 'differ'. This seemed appropriate given that algorithms such as ours are in some sense the opposite of those such as the 'diff3' algorithm, itself recently formalised [KKP]. This algorithm will flag conflicts when attempting to merge changes. On the other hand our algorithm will always merge changes without conflicts, with the result that the resultant document may appear be nonsensical in places. This trade-off means that our algorithm and ones like it are hardly suitable for version control systems, however they find a use in real-time collaborative text editors, where the nonsensical parts can immediately be edited by users.

Acknowledgements. Thank you to Jeroen Ketema for pointing out that decreasing diagrams are the best way to extend the proof of the equivalence $\tau;\rho\backslash\tau \equiv \rho;\tau\backslash\rho$ from single operations to sequences of operations.

2. Operational transformations

In this section we define our stringwise operational transformations. We do this informally first, taking two of the less obvious cases as examples, and then define them formally for each case. The main result of the section is that these definitions lead to the combined effect of any two operations executed sequentially being the same regardless of the order in which the operations are executed, provided that the second is suitably transformed relative to the first. Our operational transformations also preserve the intention of each individual operation, however we leave a proof of this until subsection 5.3.

Consider then two users making concurrent changes to a document. The first deletes four characters, the second inserts two. After applying their own operations to their document, each user applies the other's. These operations need to transformed before being

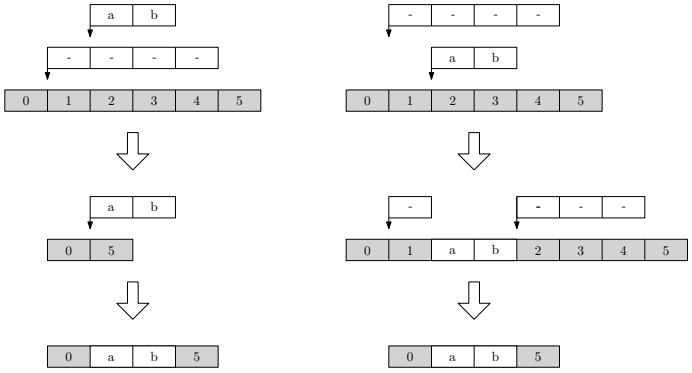


Figure 1: Operational transformations when an insert splits a delete.

applied a second time if their effect is to be preserved, however. Figure 1 illustrates the requisite transformations. On the left, the insert must be moved one character to the left. On the right, the delete must be split in two, something we consider to be unavoidable if its intention is to be preserved. The splitting of deletes in cases like this sometimes leads to an increase in the number of operations as they are transformed, a process we call fragmentation.

One way to avoid fragmentation, adopted in [RNRG96] and in the majority of attempts since, is to consider each stringwise operation as a sequence of characterwise operations which cannot be split any further. We think this is wholly impractical. Another approach, adopted in [Cor95], is not to preserve the effect of the insert at all. Figure 2 illustrates these transformations. On the left, the insert is transformed into the empty operation. On the right, the transformed delete simply deletes the inserted string. This approach also mitigates against fragmentation because the transformed delete no longer has to be split in two, but at the expense of effectively throwing away the insert or, in other words, not preserving its intention. By contrast, our approach is not to compromise and to stick with what we call a comprehensive definition of stringwise operational transformations, even if this leads to fragmentation.

The other case that deserves mention is the case when one delete splits another. Figure 3 illustrates the transformations. On the left, the first delete when transformed becomes the empty operation, since all of the characters it was to delete have been deleted already. On the right, as in the first case, the transformed delete is split in two, however the two resulting deletes lie immediately next to each other once the other delete is applied, and can therefore be treated as one operation. This result has interesting consequences, in particular proving correctness in a more general setting turns out to be impossible without it.

Now let τ and ρ be two arbitrary operations. We define $\tau\backslash\rho$ as the operation or operations that result when τ is transformed relative to ρ, and vice versa for $\rho\backslash\tau$. We can then state the combined effect of any two operations executed sequentially being the same regardless of the order in which they are executed, provided that the second is suitably transformed relative to the first, as the following equivalence:

$$\tau; \rho\backslash\tau \equiv \rho; \tau\backslash\rho \tag{2.1}$$

In the remainder of this section we give formal definitions of operational transformations for all cases, define what is meant by two operations or sequences of operations being equivalent, then prove that this equivalence always holds.

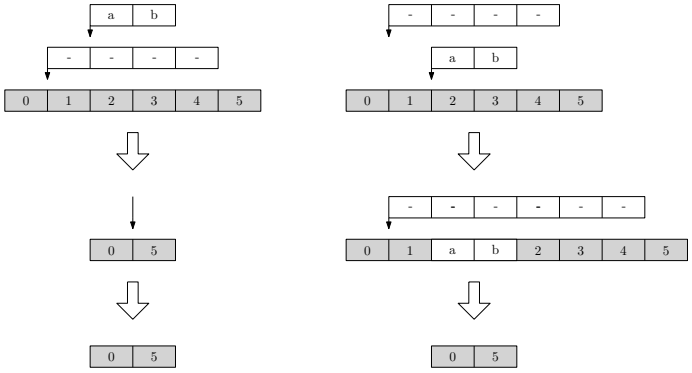


Figure 2: Less than ideal operational transformations for an insert versus a delete.

Definition 2.1. Let Σ be a non-empty, finite set of characters from some alphabet. A string is any finite sequence of characters from Σ, ranged over by s, s' and so on. The length of a string s, written $|s|$, is the length of this sequence. The set of these strings is written Σ^* and the set of non-empty strings Σ^+. We define the substring $s[n...m]$ to be the string formed by taking the n'th to the $m-1$'th characters of the string s inclusive. We also make use of the abbreviations $s[...m] = s[0...m]$ and $s[n...] = s[n...|s|]$. We write $s' + s''$ for the concatenation of strings s' and s'' in the usual sense of the word.

We define the syntax of operations as follows:

Definition 2.2. The operations τ, ρ and so on range over the following set:

$$\{i(n,s) | n \in \mathbb{N}, s \in \Sigma^+\} \cup \{d(n,l) | n \in \mathbb{N}, l \in \mathbb{N}^+\}$$

Definition 2.3. The operation ϵ ranges over the following set:

$$\{e()\}$$

Intuitively $i(n,s)$ is an insert, $d(n,l)$ a delete and $e()$ is the operation that does nothing, otherwise known as the empty operation. We say that inserts and deletes have position n.

We define the effects of operations as follows:

Definition 2.4. $i(n,s)$, $d(n,l)$ and $e()$ are partial functions, defined only for suitable strings, in which case we have:

$$\begin{aligned} i(n,s') : \Sigma^* &\longrightarrow \Sigma^+ \\ s &\longmapsto s[...n] + s' + s[n...] \end{aligned}$$

$$\begin{aligned} d(n,l) : \Sigma^+ &\longrightarrow \Sigma^* \\ s &\longmapsto s[...n] + s[n+l...] \end{aligned}$$

$$\begin{aligned} e() : \Sigma^* &\longrightarrow \Sigma^* \\ s &\longmapsto s \end{aligned}$$

By a suitable string s we mean $n \leqslant |s|$ in the case of inserts, $n+l \leqslant |s|$ in the case of deletes and any string in the case of the empty operation.

Definition 2.5. Two single operations are equivalent, that is $\tau \equiv \rho$, if and only if $\tau(s) = \rho(s)$ for any string s suitable for both τ and ρ.

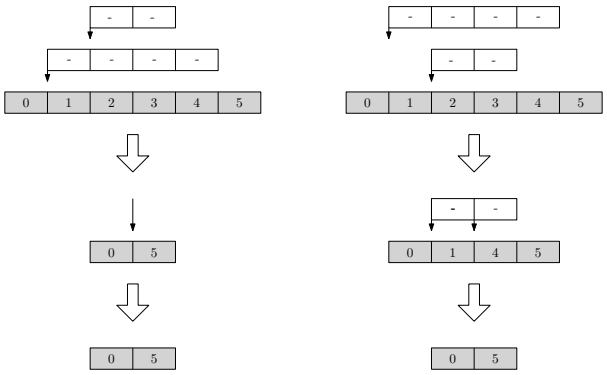


Figure 3: Operational transformations when one delete splits another.

So equivalence is defined in terms of the effect of the operations in question. It is easy to check that $\tau \equiv \rho$ precisely when τ and ρ are identical syntactically.

We next extend the notion of equivalence to sequences of operations.

Definition 2.6. Consider two sequences of operations $\tau_1; \tau_2 ...; \tau_m$ and $\rho_1; \rho_2 ...; \rho_m$. We define them as being equivalent, that is $\tau_1; \tau_2 ...; \tau_n \equiv \rho_1; \rho_2 ...; \rho_n$, if an only if $\tau_m(...\tau_2(\tau_1(s))) = \rho_n(...\rho_2(\rho_1(s)))$ for any suitable string s. By suitable we mean not only that $\tau_1(s)$ and $\rho_1(s)$ are defined, but also $\tau_2((\tau_1(s))$, $\rho_2(\rho_1(s))$ and so on.

Before continuing we make two points. The first point is that the notions of effect and equivalence here have nothing to do with the meaning of the underlying content. We hope it goes without saying that any treatment concerned with preserving meaning of this content, however this meaning might be defined, is a treatment of an entirely different problem to the one solved here. The second point is really an excuse for the definitions and results that follow. They are laborious, however a faithful implementation requires them. We nonetheless encourage the disinterested reader to move on to the next section.

Definition 2.7.

$$\begin{aligned} {}_{\llcorner}i(n,s) &= n \qquad i(n,s)_{\lrcorner} = n + |s| - 1 \\ {}_{\llcorner}d(n,l) &= n \qquad d(n,l)_{\lrcorner} = n + l - 1 \end{aligned}$$

We call ${}_{\llcorner}\tau$ and $\tau_{\lrcorner}$ the corners of τ, taking the right corner to be the position of the last character underneath the operation, so to speak. For example, in figure 4 the left and right corners of the delete are 1 and 4, respectively.

Definition 2.8.

$$\begin{aligned} \tau \ll \rho &\text{ iff } \tau_{\lrcorner} < {}_{\llcorner}\rho \\ \tau < \rho &\text{ iff } ({}_{\llcorner}\tau < {}_{\llcorner}\rho) \wedge (\tau_{\lrcorner} \geqslant {}_{\llcorner}\rho) \\ \tau \simeq \rho &\text{ iff } ({}_{\llcorner}\tau = {}_{\llcorner}\rho) \wedge (\tau \neq \rho) \\ \tau > \rho &\text{ iff } ({}_{\llcorner}\tau \leqslant \rho_{\lrcorner}) \wedge (\tau_{\lrcorner} > \rho_{\lrcorner}) \\ \tau \gg \rho &\text{ iff } {}_{\llcorner}\tau > \rho_{\lrcorner} \end{aligned}$$

These definitions formalise the idea of one non-empty operation overlapping another, regardless of whether the operations are inserts or deletes. Intuitively $\tau \simeq \rho$ when τ and ρ start in the same place but are not equal, $\tau < \rho$ when τ starts to the left of ρ but they

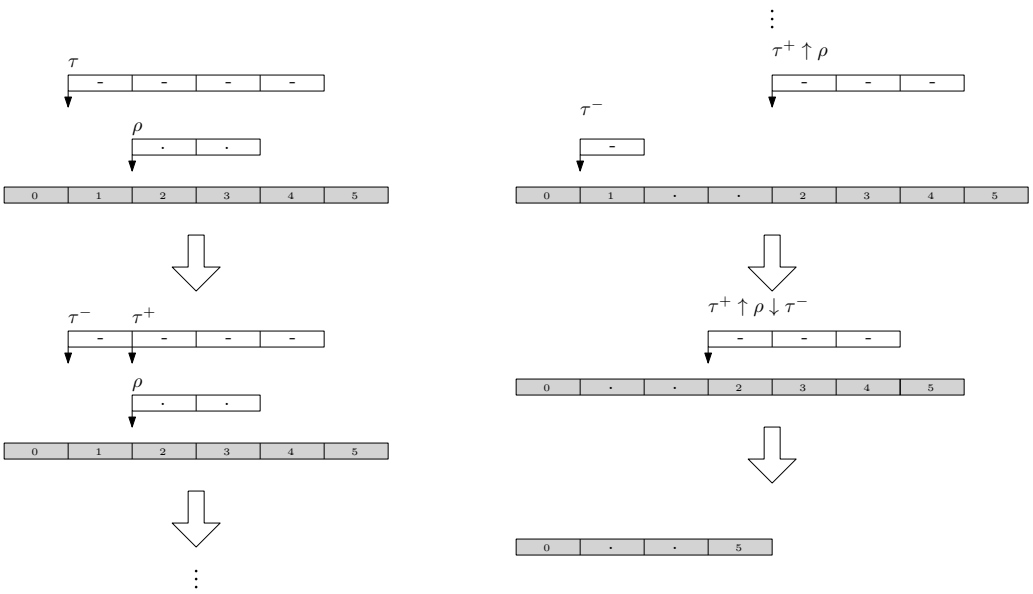


Figure 4: The reasoning behind the equivalence $\tau \backslash \rho \equiv \tau^-; \tau^+ \uparrow \rho \downarrow \tau^-$ when τ covers ρ.

overlap, and $\tau \ll \rho$ when τ starts to the left of ρ but they do not overlap. Similarly for $\tau > \rho$ and $\tau \gg \rho$.

Lemma 2.1. For any two differing operations τ and ρ, exactly one of the relations in definition 2.8 will hold. □

Next we define a series of partial transformations for inserts and deletes that formalise the idea of one non-empty operation being shifted one way or the other by another.

Definition 2.9. For two inserts $i(n_1, s_1)$ and $i(n_2, s_2)$ with $n_1 \geqslant n_2$:

$$i(n_1, s_1) \uparrow i(n_2, s_2) = i(n_1 + |s_2|, s_1).$$

For a delete $d(n_1, l_1)$ and insert $i(n_2, s_2)$ with $n_1 \geqslant n_2$:

$$d(n_1, l_1) \uparrow i(n_2, s_2) = d(n_1 + |s_2|, l_1).$$

Definition 2.10. For two deletes $d(n_1, l_1)$ and $d(n_2, l_2)$ with $n_1 \geqslant n_2 + l_2$:

$$d(n_1, l_1) \downarrow d(n_2, l_2) = d(n_1 - l_2, l_1)$$

For an insert $i(n_1, s_1)$ and delete $d(n_2, l_2)$ with $n_1 \geqslant n_2 + l_2$:

$$i(n_1, s_1) \downarrow d(n_2, l_2) = i(n_1 - l_2, s_1)$$

Intuitively $\tau \uparrow \rho$ is τ shifted to the right by the length of ρ when ρ is an insert; and $\tau \downarrow \rho$ is τ shifted to the left by the length of ρ when ρ is a delete. Note the restrictions on the relative positions of the operations in each case. There is never a need to shift an operation to the right by the length of an insert if that operation is already to its left. Similarly there is never a need to shift an operation to the left by the length of a delete unless that operation is to its right.

Finally we define partial transformations that split or crop one non-empty operation relative to another. The motivation for these can be seen in figure 4 again. Compare this with figure 1, where the various steps involved in transforming the delete relative to the

insert were left to the imagination. Figure 4 on the other hand makes these steps explicit. The delete is first subdivided, and then the right side must be further shifted twice. The result, as expected, is that the transformed delete operation will delete the same characters, albeit either side of the inserted characters, that the original delete operation would have deleted were the insert operation not to be applied first. Its intention is preserved, in other words.

Definition 2.11. For two deletes $d(n_1, l_1)$ and $d(n_2, l_2)$:

$$d(n_1,l_1) - d(n_2,l_2) = \begin{cases} d(n_1, n_2 - n_1) & n_1 < n_2 \qquad n_2 < n_1 + l_1 \leqslant n_2 + l_2 \\ d(n_2 + l_2, n_1 + l_1 - n_2 - l_2) & n_1 + l_1 > n_2 + l_2 \qquad n_2 \leqslant n_1 < n_2 + l_2 \end{cases}$$

Intuitively if τ overlaps ρ either to the left or the right, then $\tau - \rho$ is τ with that part overlapping with ρ chopped off.

Definition 2.12. For a delete $d(n_1, l_1)$ and an insert $i(n_2, s_2)$:

$$\left.\begin{aligned} d(n_1,l_1)^- &= d(n_1, n_2 - n_1) \\ d(n_1,l_1)^+ &= d(n_2, l_1 - n_2 + n_1) \end{aligned}\right\} \; n_1 < n_2 \quad n_1 + l_1 > n_2 + |s_2|$$

Definition 2.13. For two deletes $d(n_1, l_1)$ and $d(n_2, l_2)$:

$$\left.\begin{aligned} d(n_1,l_1)^- &= d(n_1, n_2 - n_1) \\ d(n_1,l_1)^+ &= d(n_2 + l_2, n_1 + l_1 - n_2 - l_2) \end{aligned}\right\} \; n_1 < n_2 \quad n_1 + l_1 > n_2 + l_2$$

Intuitively if τ covers ρ, then ρ splits τ into τ^- and τ^+. If ρ is an insert, the split takes place at the position of ρ and none of τ is lost. If ρ is a delete, only the parts of τ on either side of ρ are kept. Note that we drop any reference to ρ in these definitions, but it is always clear what ρ is from the context.

We are now in a position to prove the main result of this section.

Theorem 2.1. For any two single operations τ and ρ and for suitable definitions of the transformed operations $\tau\backslash\rho$ and $\rho\backslash\tau$, equivalence 2.1 holds.

Proof. We break the proof down into cases.

When τ and ρ are both inserts, we set $\tau = i(n_1, s_1)$, $\rho = i(n_2, s_2)$. Without loss of generality suppose that $n_1 \leqslant n_2$. We treat the case when $n_1 < n_2$ first, in which case we set $i(n_1, s_1)\backslash i(n_2, s_2) = i(n_1, s_1)$ and $i(n_2, s_2)\backslash i(n_1, s_1) = i(n_2, s_2) \uparrow i(n_1, s_1)$. For any suitable s we then have:

$$\begin{aligned} (i(n_1,s_1); i(n_2,s_2)\backslash i(n_1,s_1))(s) &= (i(n_1,s_1); i(n_2 + |s_1|, s_2))(s) \\ &= i(n_2 + |s_1|, s_2)(s[...n_1] + s_1 + s[n_1...]) \\ &= s[...n_1] + s_1 + s[n_1...n_2] + s_2 + s[n_2...] \\ &= i(n_1,s_1)(s[...n_2] + s_2 + s[n_2...]) \\ &= (i(n_2,s_2); i(n_1,s_1))(s) \\ &= (i(n_2,s_2); i(n_1,s_1)\backslash i(n_2,s_2))(s) \end{aligned}$$

For the sake of the reader who has gotten this far we omit similar proofs from now on.

To continue, the case when $n_1 = n_2$ is more subtle. We assume that there is some lexicographical ordering on the strings, that is for two strings $s_1, s_2 \in \Sigma^*$ we have $s_1 < s_2$, $s_1 = s_2$ or $s_1 > s_2$. When $s_1 = s_2$ the transformations need do nothing for equivalence 2.1 to hold.

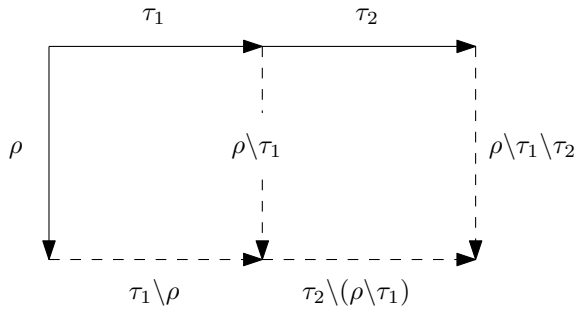


Figure 5: Transforming a sequence of operations $\tau = \tau_1;\tau_2$ relative to a single operation ρ.

Without loss of generality suppose now that $s_1 < s_2$ and set $i(n_1, s_1)\backslash i(n_2, s_2) = i(n_1, s_1)$ and $i(n_2, s_2)\backslash i(n_1, s_1) = i(n_2 + |s_1|, s_2)$. These transformations result in the lexicographically lesser of the two operations remaining in place whilst the other is shifted to the right, regardless of the order of application, and so again equivalence 2.1 holds. Hence in all the cases when both τ and ρ are inserts, equivalence 2.1 holds.

From now on we just state the transformations in each case, leaving the proofs to interested readers. These are easily done along the lines of figure 4.

When τ and ρ are both deletes, we only need to consider the cases when $\tau = \rho$, $\tau \simeq \rho$ with ${}_{\llcorner}\tau < {}_{\llcorner}\rho$, $\tau < \rho$ with $\tau_{\lrcorner} < \rho_{\lrcorner}$, $\tau < \rho$ with $\tau_{\lrcorner} = \rho_{\lrcorner}$, $\tau < \rho$ with $\tau_{\lrcorner} > \rho_{\lrcorner}$, or $\tau \ll \rho$. Symmetry takes care of the remaining cases. In the case when $\tau = \rho$, it suffices to set $\tau\backslash\rho = \rho\backslash\tau = \epsilon$. In the case when $\tau \ll \rho$, it suffices to set $\tau\backslash\rho = \tau$ and $\rho\backslash\tau = \rho \downarrow \tau$. The other cases are a little more involved. In the case when $\tau \simeq \rho$ and $\tau_{\lrcorner} < \rho_{\lrcorner}$ we set $\tau\backslash\rho = \epsilon$ and $\rho\backslash\tau = (\rho - \tau) \downarrow \tau$. In the case when $\tau < \rho$ with $\tau_{\lrcorner} < \rho_{\lrcorner}$ we set $\tau\backslash\rho = \tau - \rho$ and $\rho\backslash\tau = (\rho - \tau) \downarrow \tau$ again. In the case when $\tau < \rho$ with $\tau_{\lrcorner} = \rho_{\lrcorner}$ we set $\tau\backslash\rho = \tau - \rho$ and $\rho\backslash\tau = \epsilon$. Only the case when τ covers ρ remains, namely when $\tau < \rho$ with $\tau_{\lrcorner} > \rho_{\lrcorner}$. Here we set $\tau\backslash\rho = \tau^-;\tau^+ \downarrow \rho \downarrow \tau^-$ and $\rho\backslash\tau = \epsilon$, noting again that the first of these transformations results in a single transformed operation, not two. Hence in all the cases when both τ and ρ are deletes, equivalence 2.1 holds.

When τ is a delete and ρ an insert, we need to consider the cases $\tau \gg \rho$, $\tau > \rho$, $\tau \simeq \rho$, $\tau < \rho$, or $\tau \ll \rho$. The relative positions of the right corners of the operations do not come into account. In the cases when $\tau \gg \rho$, $\tau > \rho$, or $\tau \simeq \rho$ we set $\tau\backslash\rho = \tau \uparrow \rho$ and $\rho\backslash\tau = \rho$. In the case when $\tau < \rho$ we set $\tau\backslash\rho = \tau^-;\tau^+ \uparrow \rho \downarrow \tau^-$, as illustrated in figure 4, and $\rho\backslash\tau = \rho \downarrow \tau^-$. Finally, in the case when $\tau \ll \rho$ we set $\tau\backslash\rho = \tau$ and $\rho\backslash\tau = \rho \downarrow \tau$. Hence in all the cases when τ is a delete and ρ an insert, equivalence 2.1 holds.

The trivial observation that if τ is an insert and ρ a delete we simply swap the symbols above completes the proof. □

3. Decreasing diagrams

In this section we prove that equivalence 2.1 can be extended to the cases when τ and ρ each represent sequences of operations rather than just one. We then use this result to devise a method that transforms one sequence of operations relative to another.

To begin with we consider the simplest case of when τ is a sequence of two operations $\tau_1;\tau_2$ whilst ρ remains a single operation. Figure 5 shows a representation of this case that turns out to be useful. Here solid edges represent single operations, dashed edges possibly

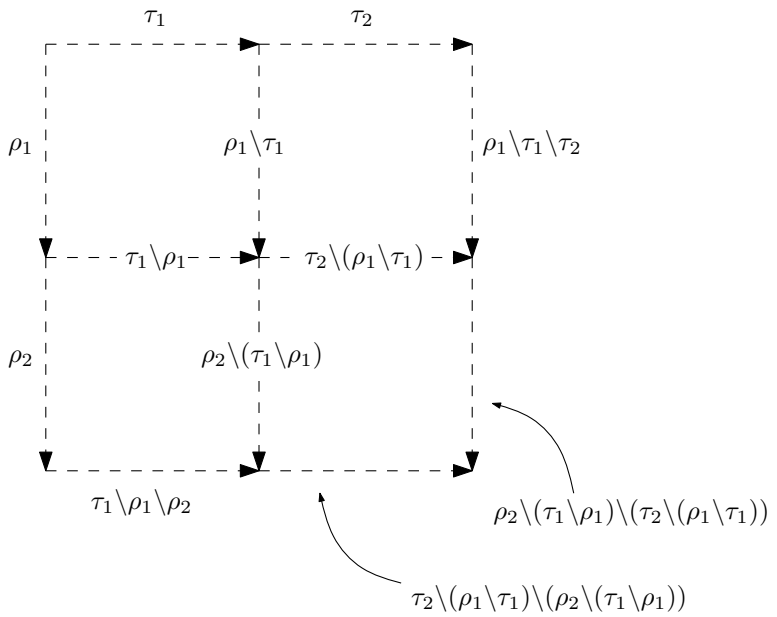


Figure 6: Tranforming one sequences of operations $\tau = \tau_1; \tau_2$ relative to another $\rho = \rho_1; \rho_2$.

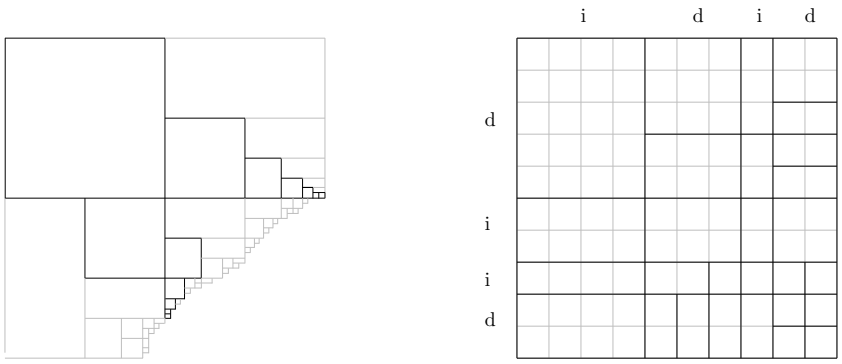
two. From theorem 2.1 we know that the left hand side of this representation commutes. However we cannot as yet prove that the right hand side commutes, because $\rho\backslash\tau_1$ is not necessarily a single operation. Therefore we cannot prove that the representation as a whole commutes.

Now we go on to consider the general case when both τ and ρ are sequences of operations of arbitrary length, split in two. Thus τ becomes $\tau_1; \tau_2$ and ρ becomes $\rho_1; \rho_2$. Figure 6 shows the representation. Note that all the edges are dashed, since no assumptions are made about the lengths of the sub-sequences. At this point we clearly have no way of proving that any part let alone the whole of this representation commutes. Nonetheless, bearing in mind that τ_1, τ_2, ρ_1 and ρ_2 are just labels, then if the representation commutes it should be possible to extend equivalence 2.1 to the following:

$$\tau_1; \tau_2; (\rho_1; \rho_2)\backslash(\tau_1; \tau_2) \equiv \rho_1; \rho_2; (\tau_1; \tau_2)\backslash(\rho_1; \rho_2) \tag{3.1}$$

In order to prove this equivalence we put it to one side for now and streamline the representations. We do away with the arrows, since the direction is always clear. We do away with naming the edges at all, in fact, although we label them as being deletes or inserts. Since an insert is still an insert when transformed, and a delete is also still a delete or possibly two deletes when transformed, we only need to label the topmost and leftmost edges. We also do away with dashed edges, splitting the edges instead when necessary. There are two slight caveats. The first is that a delete may end up being the empty operation when transformed relative to another delete. We therefore introduce deletes of length zero at this stage to avoid having to re-label the edges. The second is that deletes are not necessarily split by inserts, however we assume that they always are and thus cover the worst case scenario. We call the resultant streamlined representations diagrams, after [KvOdV00].

In fact our diagrams represent abstract rewriting systems and the property of our earlier representations being commutative can be restated as these diagrams being Church-Rosser. Diagrams with topmost and leftmost edges representing single operations are called

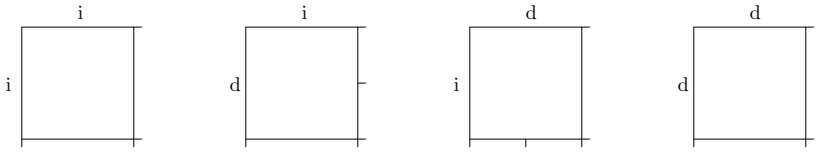


(a) (b)

Figure 7: Infinite sequences of sub-diagrams and the "squared paper" argument.

elementary, and it turns out that we have already proved that they are Church-Rosser. All we need do is to re-state theorem 2.1 diagrammatically, so to speak.

Theorem 3.1. The following elementary diagrams are Church-Rosser:



□

If all the elementary diagrams contained in a diagram are Church-Rosser, then the diagram is said to be weakly Church-Rosser. Furthermore Newman's lemma [New42] in this context means that if a weakly Church-Rosser diagram contains no infinte sequences of elementary diagrams, in which case it is known as terminating, then it is Church-Rosser. Figure 7 (a) illustrates a somewhat unlikely, partially drawn diagram containing infinite sequences of elementary diagrams that is weakly Church-Rosser. The labels have been omitted, indeed no labelling would make sense. For example, the right-most and bottom-most edges of the largest elementary diagram are both split, and as such it does not match one of the four available elementary diagrams listed in theorem 3.1. A little experimentation with these should convince that it is impossible to construct such infinite sequences. What we do next is make this intuition precise.

It turns out that we are already in a position to show that these diagrams are Church-Rosser. We simply note that we are dealing with discrete data that cannot be split indefinitely. If we imagine the length of the topmost and leftmost edges of any elementary diagram to be proportional to the length of operations they represent, and then divide the bottom-most and right-most edges accordingly if one of these operations is a delete that has been split or shortened when transformed, the resulting diagrams could be drawn on squared paper with no elementary diagram ever being smaller than a square. Figure 7 (b) illustrates an example of this argument, which perhaps could be stated a little more fully and precisely, but really does not need to be. By this argument we have:

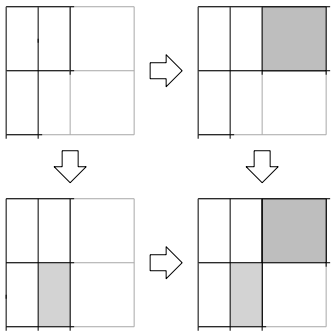


Figure 8: The choice of two elementary diagrams commutes.

Theorem 3.2. Equivalence 2.1 holds in the case when the data is discrete and τ and ρ each represent sequences of operations rather than just one. □

Next, we prove that our diagrams are Church-Rosser in what could be called continuous case, that is without any recourse to the fact that the data is discrete. The justification for this is that the more painstaking arguments involved lead to insights and besides, have an intrinsic value of their own.

Before proceeding we make one further observation, namely that our proof will require that diagrams are filled in in a particular way, which we go on to define, whilst it is possible that they could be filled in in many different ways. In [KvOdV00] the authors acknowledge this fact, but claim that it is not hard to see that the same diagram results regardless of the way in which it is filled in. As justification for this they show that if there is a choice of two elementary diagrams to be filled in at any stage, then the order of the choices commutes, as illustrated in figure 8. Whilst this fact is evidently true, we feel that the argument itself deserves a closer look. In the case of infinite diagrams, for example, it is possible to fill in an infinite sequence of elementary diagrams without ever reaching other parts of the diagram. Whatever the case, we define a systematic way of filling in what we call maximal sub-diagrams. We then define a unique maximal diagram and show that any diagram is a subset of this. We are then free to fill in diagrams in any way we choose.

Lemma 3.1. For any two sequences of non-empty operations there exists a nested sequence of unique maximal sub-diagrams.

Proof. Given the two sequences we construct a grid, labelling its edges according to the types of the operations. To fill in the grid we define a simple algorithm. We place a pointer at the bottom, right hand corner of the grid. If we cannot move it to the left, we are done. Otherwise we move it to the left until we encounter either the right-most edge of an elementary diagram or the leftmost edge of the grid. Next we move it up until we encounter the bottom-most edge of an elementary diagram or the topmost edge of the grid. When we cannot move the pointer up any further we have reached what we call a corner. Each corner is bounded to the left and on top by existing elementary diagrams or edges of the grid and we therefore have but one choice in filling in the next elementary diagram. Once done, we place the pointer at the top, right corner of the elementary diagram we have just filled in. If it is a corner, we fill in another elementary diagram and move the pointer as before. When we cannot fill in any more corners, either we can move the pointer up in search of the next corner, or we have reached the right-most edge of the grid, in which case we have completed

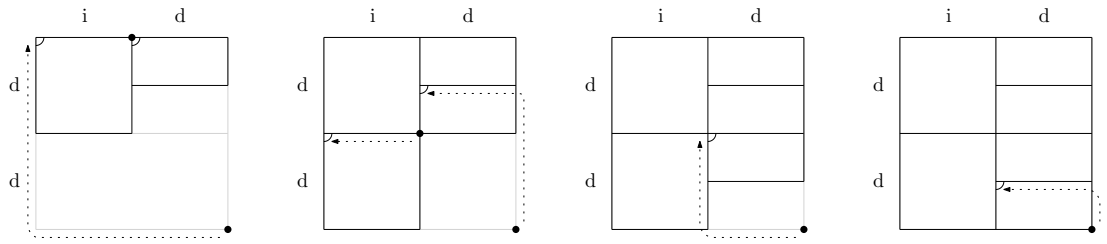


Figure 9: The stages of a simple diagram being filled in.

what we call a stage. We then place the pointer at the bottom, right hand corner of the grid again, this time we working up initially and then to the left and down, filling in corners as we go until we have completed a stage in the other direction. We then repeat these stages alternately, creating a nested sequence of unique maximal sub-diagrams. □

Figure 9 illustrates an example of this algorithm in progress. Solid discs represent the position of the pointer at the beginning of each stage, quarter circles represent corners against the previous stage's maximal sub-diagram. These are filled in with elementary diagrams to make the stage's maximal sub-diagram. We say these elementary diagrams border the previous maximal sub-diagram. The algorithm may run indefinitely. In figure 7 (a), for example, it appears to have completed roughly six stages.

Implicit in all of this is is our definition of a diagram:

Definition 3.1. A diagram is a sequence of elementary diagrams each of which fills in a corner.

So we define diagrams a sequences of elementary diagrams rather than just sets. Note that two distinct diagrams may have the same set of elementary diagrams, but filled in in a different order. Note also that we will abuse the definition a little in what follows, talking of one diagram being the subset of another diagram if its corresponding set of elementary diagrams is a subset of the other's. In fact we have abused it already in talking of a nested sequence of maximal sub-diagrams. Moving swiftly on, we define what we call a unique maximal diagram and then give the lemmas we need:

Definition 3.2. The maximal diagram is the unique sequence of elementary diagrams generated by the algorithm, whether or not it terminates.

Lemma 3.2. Any maximal sub-diagram is a subset of the maximal diagram. □

Lemma 3.3. Any finite diagram is the subset of a maximal sub-diagram.

Proof. By complete induction on the sequence. Since the first elementary diagram of this sequence always occupies the top, left corner of the grid, it must be the same as the first elementary diagram of the first maximal sub-diagram and the base case is proved. Now consider the $k+1$'th elementary diagram of the sequence. Our induction hypothesis is that the diagram consisting of the first k elementary diagrams is the subset of say the l'th maximal sub-diagram. Then there are two cases. Either the $k+1$'th elementary diagram is in the l'th maximal sub-diagram in which case we are done, or it is not. If not, it fills in a corner that borders the l'th maximal sub-diagram and therefore must be in the $l+1$'th maximal sub-diagram and again we are done. □

Lemma 3.4. Any finite diagram is a subset of the maximal diagram.

Proof. This follows from lemmas 3.2 and 3.3. □

So we are free to fill in diagrams in any way we choose. Now we build up to the main result of this section.

Definition 3.3. A row is a diagram with one operation along its leftmost edge. It is said to start with this edge. It's length is the number of operations along its topmost edge. Similarly for columns.

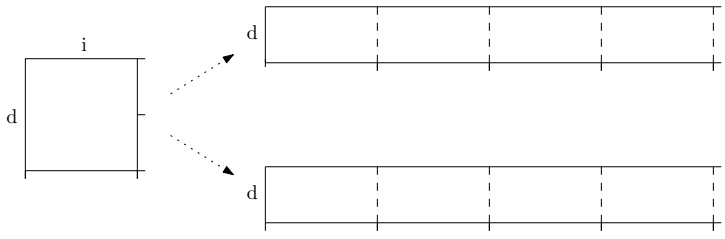
Lemma 3.5. The number of operations along the bottom-most edge of a row starting with an insert is at most double the number of operations along its topmost edge. Similarly for columns.

Proof. By induction on the length of the row. The base case is easy. Now consider a row of length $k+1$ starting with an insert. It can only start with one of two elementary diagrams, one of which has one operation along its bottom-most edge, the other two:

By the induction hypothesis the number of operations along the bottom-most edge of the remaining row of length k is at most double the number of operations along its topmost edge and thereore the number of operations along the bottom-most edge of the whole row of length $k+1$ is at most double the number of operations along its topmost edge. □

Lemma 3.6. The number of operations along the topmost and bottom-most edges of a row starting with a delete is equal. Similarly for columns.

Proof. By induction on the length of the row. The base case is easy. Now consider a row of length $k+1$ starting with a delete. If the first topmost operation is a delete, the opposite edge to the starting edge of the first elementary diagram is a single delete and we can apply the induction hypothesis to the remaining row of length k. If on the other hand the first topmost operation is an insert, the opposite edge to the starting edge of the first elementary diagram consists of two deletes:



We apply the induction hypothesis carefully. The row starting with the first of these two deletes has k operations along its topmost edge. By the induction hypothesis it has k operations along its bottom-most edge. This bottom-most edge is the topmost edge of the other row. By the induction hypothesis this also has k operations along its bottom-most edge. Therefore the whole row has $k+1$ operations along its bottom-most edge. □

There are several ways to proceed from here. We simply prove that the number of operations along the edges of the diagram remains finite. Or to put it another way, the number of operations no more than doubles with each row or column.

Lemma 3.7. A diagram with R rows and n_R operations along its topmost edge has at most $2^R \times n_R$ operations along its bottom-most edge.

Proof. By induction on the number of rows and by lemmas 3.5 and 3.6. □

Lemma 3.8. A diagram with C columns and n_C operations along its leftmost edge has at most $2^C \times n_C$ operations along its right-most edge.

Proof. By induction on the number of columns and by lemmas 3.5 and 3.6. □

Definition 3.4. A diagram is terminating if it contains no infinite sub-sequences of elementary diagrams.

Observation 3.1. A diagram is terminating if and only if it has a finite number of operations along its right-most and bottom-most edges.

Corollary 3.1. Our diagrams are terminating.

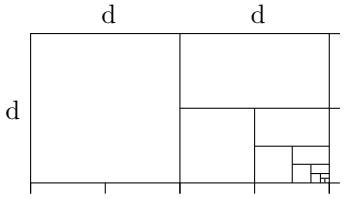
Proof. A result of lemma 3.7, lemma 3.8 and the previous observation. □

Theorem 3.3. Our diagrams are Church-Rosser.

Proof. Our diagrams are locally Church-Rosser by theorem 3.1, terminating by corollary 3.1 and therefore, by Newman's lemma, Church-Rosser. □

Interestingly, we cannot do without the fact that deletes do not split deletes:

Counter-example 3.1. If a delete splits a delete, our diagrams are not Church-Rosser.



What use is this fancy formalism? What it means is that given sequences of operations τ and ρ, the transformation $\tau \backslash \rho$ can be computed. To see why, we note that the existence of the elementary diagrams in theorem 3.1 asserts a priori that their bottommost and rightmost edges can be computed. Since our diagrams always exist regardless of the combinations of inserts and deletes in each sequence, the transformations can be computed. To see how, based on the representation in figure 6 we suggest the following reductions:

$$\rho \backslash (\tau_1 ; \tau_2) \rightsquigarrow \rho \backslash \tau_1 \backslash \tau_2$$
$$(\tau_1 ; \tau_2) \backslash \rho \rightsquigarrow \tau_1 \backslash \rho ; \tau_2 \backslash (\rho \backslash \tau_1)$$
$$(\tau_1 ; \tau_2) \backslash (\rho_1 ; \rho_2) \rightsquigarrow \tau_1 \backslash \rho_1 \backslash \rho_2 ; \tau_2 \backslash (\rho_1 \backslash \tau_1) \backslash (\rho_2 \backslash (\tau_1 \backslash \rho_1))$$

We now outline a method by way of an example. Suppose that τ_1, τ_2 and ρ are single operations and we wish to transform ρ relative to $\tau_1 ; \tau_2$. With the above reductions to hand

we give a series of steps that reduce $\rho\backslash(\tau_1;\tau_2)$ to sequence of operations:

$$\begin{aligned}\rho\backslash(\tau_1;\tau_2) &\rightsquigarrow \rho\backslash\tau_1\backslash\tau_2\\ &= (\rho';\rho'')\backslash\tau_2\\ &\rightsquigarrow \rho'\backslash\tau_2;\rho''\backslash(\tau_2\backslash\rho')\end{aligned}$$

We walk through the steps. To begin with we employ the first reduction. Then since ρ and τ_1 are single operations we can compute $\rho\backslash\tau_1$. Let us say it becomes $\rho';\rho''$ where ρ' and ρ'' are single operations. Next we employ the second reduction and then, since ρ' and τ_2 are single operations we can compute $\rho'\backslash\tau_2$, which we leave as is. Next we note that ρ' must be a delete and therefore $\tau_2\backslash\rho'$ must be a single operation, therefore we can also compute $\rho''\backslash(\tau_2\backslash\rho')$ and we are done.

Our method therefore consists of a single recursive function that takes two sequences of operations, employing the requisite reductions in order to break down these sequences into sub-sequences, then calling itself passing in these new sequences. When the length of both sequences is one, it can then transform the operation in the first sequence relative to the operation in the second sequence. The fact that our diagrams are finite tells us that this process must eventually terminate. Furthermore it does not matter how we choose to break up any given sequence although in practice we hive off the first operation from any sequence of operations. An interesting and as yet unanswered question is whether breaking the sequences roughly in the middle, say, is more efficient. Certainly the function as it stands seems to be fast enough. Transformations involving sequences of hundreds of operations are computed in virtually no time.

Given that the right hand sides of the reductions can be computed we are free to couch them as identities, which will be utilised in the next section:

$$\rho\backslash(\tau_1;\tau_2) = \rho\backslash\tau_1\backslash\tau_2 \tag{3.2}$$

$$(\tau_1;\tau_2)\backslash\rho = \tau_1\backslash\rho;\tau_2\backslash(\rho\backslash\tau_1) \tag{3.3}$$

$$(\tau_1;\tau_2)\backslash(\rho_1;\rho_2) = \tau_1\backslash\rho_1\backslash\rho_2;\tau_2\backslash(\rho_1\backslash\tau_1)\backslash(\rho_2\backslash(\tau_1\backslash\rho_1)) \tag{3.4}$$

Note that these are indeed identities, not equivalences. They say nothing about the effects on any string s. Nonetheless given that for sequences of operations τ and ρ the transformations $\tau\backslash\rho$ and $\rho\backslash\tau$ can be computed and given that the representation in figure 6 commutes, we do have the following:

Theorem 3.4. Equivalence 3.1 holds. □

In other words equivalence 2.1 can be extended to the cases when τ and ρ each represent sequences of operations rather than just one, which is what we set out to prove.

4. The protocol

In this section we devise an algorithm that is able to keep any number of copies of a document in line whilst changes are made to them concurrently. In order to do so it utilises the method outlined in the previous section to transform one sequence of operations relative to another, together with a simple protocol which sits on top of the HTTP protocol. Since the HTTP protocol is based on the client-server model, so is our algorithm. Crucially, the choice of a client-server model leads to the correct application of operational transformations to both newly generated *and* pending operations. We explain what we mean by this later.

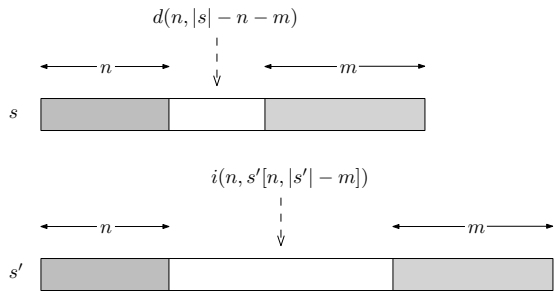


Figure 10: Generating operations from a comparison of strings s and s'.

Just as in the previous sections, the treatment here is predominantly theoretical although we draw parallels with the workings of the implementation where appropriate. One instance where practical considerations had a bearing on the theory was the problem of latency. Initially this was overlooked, and then when issues arose, they had to be addressed. In the end the solution required no more than a refinement of the protocol.

We therefore break this section into two. In the first part, we adopt a woolly notion of global time and present a protocol based on that, neglecting the problem of latency. This allows us to get the salient points across. In the second part, we do away with global time and adopt Lamport's "happens before" relation [Lam78]. This provides the correct context in which to explain the solution to the problem of latency, namely a refined protocol, and to give a general proof.

Before getting going we briefly describe how stringwise operations are generated from a comparison of two differing strings. Such a comparison results in at most one delete and one insert operation. Figure 10 illustrates this. Here the shaded parts of strings s and s' are identical front and back. The middle parts, if there are any, contribute to a delete operation in the case of s and an insert operation in the case of s'. It is easy to check the following:

$$s' = (d(n, |s| - n - m); i(n, s'[n, |s'| - m]))(s)$$

Note that it makes no difference whether the difference between the strings is one character or many, still at most two operations are generated. This is in marked contrast to characterwise operations, where copying a whole block of text into the input field, for example, might result in an overwhelming number of operations. The clear advantage of stringwise operations should be self-evident here.

To begin the first part of this section proper we note that the client-server model consists of any number of clients, we kick off with two for illustrative purposes, and a single server. Here each client has a copy of the document, as does the server, which also has a store of pending operations for each client. Communication is carried out by way of transactions, with each transaction consisting of two parts: a request from the client, which garners a response from the server. A request consists of a command and an optional sequence of operations. A response consists of either copies of the document or sequences of operations. The protocol consists of just three types of transaction, summarised as follows:

- INITIALISE: the server responds with a copy of its document,
- PUT: the client puts a sequence of operations on the server, the server confirms,
- GET: the client requests its pending operations, and the server duly responds.

Figure 11 illustrates the likely first few transactions of a session. Here the first client is initialised with document s' and then puts a sequence of operations τ' on the server,

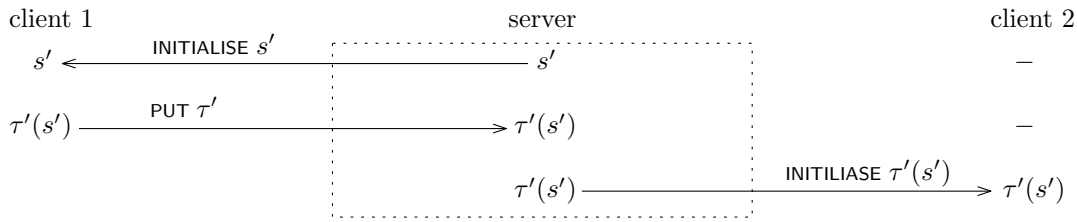


Figure 11: Two INITIALISE transactions with one PUT transaction in-between

possibly over the course of several transactions. The server applies these operations to its own copy of the document and when a second client is initialised, it receives this amended document $\tau'(s')$. In this simple case both clients end up with copies of the document that are in line with the server's copy.

Before going any further we describe these illustrations in detail and more importantly the assumptions inherent in them. We assume that the server was initialised at some point, before any of the clients. We also assume that transactions are completed, by which we mean that requests always garner responses. This cannot be guaranteed, of course, however fault tolerance can be built in for the occasions when transactions fail. See the end of the related work and conclusions section for the details. Moving on, since information only ever flows in one direction we do not show requests and responses as separate arrows, instead showing each transaction as an arrow in the appropriate direction and labelled with the requisite information. One assumption that we do not have to make is that transactions are handled sequentially by both the clients and the server. This can in fact be guaranteed in the implementation, and it means that the arrows in these illustrations never meet and never cross. We therefore draw them horizontally, assuming that transactions happen instantaneously and time unfolds incrementally. This is our woolly notion of global time.

Figure 12 illustrates the continuation of the session, with both clients having the document $\tau'(s') = s$. The first client now puts another sequence of operations τ on the server, again possibly over the course of several transactions, and this time the server stores these for the second client. Thus the first client's operations τ become the second client's pending operations and the server's document becomes $\tau(s)$. Then the second client puts its own sequence of operations ρ on the server and here come the crucial steps:

the second client's operations do not become the first client's pending operations without first being transformed relative to second client's own pending operations

Thus the first client's pending operations are $\rho\backslash\tau$ and not just ρ.

the server does not apply the second client's operations to its document without first transforming them relative to the second client's pending operations

So the server applies the operations $\rho\backslash\tau$ to its document, which becomes $(\tau; \rho\backslash\tau)(s)$.

the second client's pending operations are also transformed relative to the operations it has just generated

Therefore the second client's pending operations τ become $\tau\backslash\rho$.

In this case both clients again end up with copies of the document that are in line with the server's copy, if we assume that $(\tau; \rho\backslash\tau)(s) = (\rho; \tau\backslash\rho)(s)$, an assumption based on

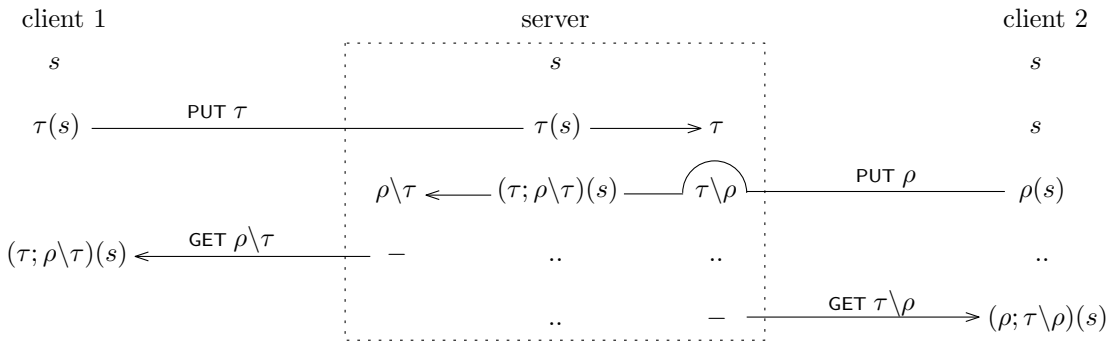


Figure 12: Two PUT transactions followed by two GET transactions

theorem 3.4. Next we clear up a technicality used in these arguments, namely that it makes no difference whether a sequence of operations is put on the server by a particular client over the course of several transactions or just one:

Lemma 4.1. Suppose the second client puts a sequence of operations ρ on the server over the course of n transactions, thus $\rho = \rho_1;\rho_2...;\rho_n$. Then its pending operations, if τ beforehand, become $\tau\backslash\rho$; the server's document, if $\tau(s)$ beforehand, becomes $(\tau;\rho\backslash\tau)(s)$; and the first client's pending operations, if the empty sequence beforehand, become $\rho\backslash\tau$.

Proof. By finite induction on the number of transactions. The base case is given by the steps above. Now suppose that the first $k-1$ transactions have taken place and set $\rho' = \rho_1;\rho_2...\rho_{k-1}$. By the induction hypothesis the second client's pending operations are $\tau\backslash\rho'$, the server's document $(\tau;\rho'\backslash\tau)(s)$ and the first client's pending operations $\rho'\backslash\tau$. Now suppose the client puts the next sequence of operations ρ_k on the server. Then, for the second client's pending operations we have, by identity 3.2:

$$(\tau\backslash\rho')\backslash\rho_k = \tau\backslash(\rho';\rho_k)$$

For the server's document, by identity 3.3:

$$(\tau;\rho'\backslash\tau;\rho_k\backslash(\tau\backslash\rho'))(s) = (\tau;(\rho';\rho_k)\backslash\tau)(s)$$

And for the first client's pending operations, again by identity 3.2:

$$\rho'\backslash\tau;\rho_k\backslash(\tau\backslash\rho') = (\rho';\rho_k)\backslash\tau$$

This completes the proof. □

It should be clear that these arguments can be generalised but we leave off doing so until we have a proper notion of time. Nonetheless it is worth pointing out that the crux is here. Sequences of pending operations for each client must be held on the server because the server cannot pass them on to each client immediately. The HTTP protocol typically does not allow information to be pushed, only pulled, and so we simulated the pushing of information by having each client poll the server, a common practice. And the storage of pending operations on the server led in turn to them being transformed in the aforementioned symmetric way.

We now come to the second part of this section and to an explanation of the solution to the problem of latency together with a general proof. Because there is no direct user interaction with the server and because it only handles one transaction at a time, the assumptions we have made about it thus far remain valid and, in particular, the storage

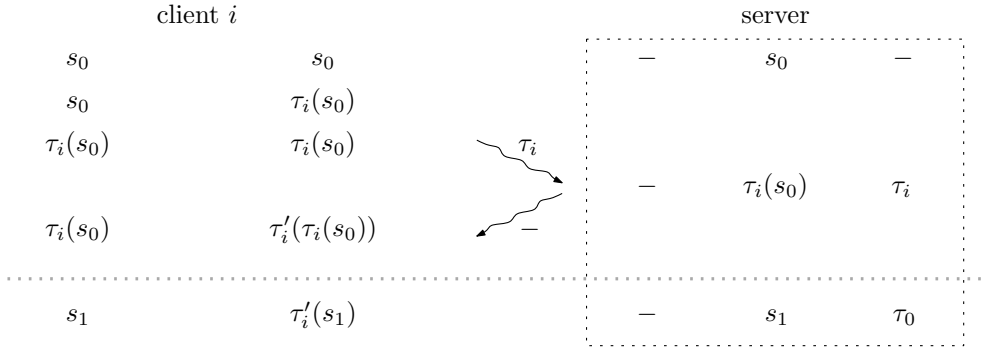


Figure 13: The first UPDATE transaction in the general case

and transformation of operations on the server does not need to change. On the other hand the state of a client may change due to user interaction whilst a transaction is in progress and this has not taken into account. We therefore do so now, bringing the treatment more into line with the implementation as it stands. This implementation includes a refined protocol consisting of just two types of transaction, summarised as follows:

- INITIALISE: the server responds with a copy of its document,
- UPDATE: the client puts a sequence of operations on the server, and the server responds with the client's pending operations, suitably transformed.

Next we introduce the fact that clients keep not one copy of the document but two. The first is a working copy, the one formalised thus far, whilst the second is an editable copy considered to be the value of the input field made available to the user. Also from now on we work with an arbitrary, albeit fixed, number of clients, rather than just two. We represent the client involved in a particular transaction at any time as the i'th client, whilst any other client we represent as the j'th client.

Figure 13 illustrates the first UPDATE transaction of a session with this new protocol, assuming that each client has already completed an INITIALISE transaction. We describe these illustrations in detail again first. The far left column represents the i'th client's working copy of the document, the column next to this its editable copy. The server's columns are the same as before. We neglect to show the j'th client this time because it is not actively involved in the i'th client's transaction. Time unfolds top to bottom as before, but only does so during the course of a transaction. The topmost and bottommost lines show information across both the i'th client's and server's columns seemingly at the same time, but this only represents the fact that both have a state before and after the transaction. It is not a return to the woolly notion of global time.

Now suppose that a user makes a change to the i'th client's editable copy of the document. The i'th client duly updates its working copy and computes the requisite operations τ_i, sending these to the server. Since this is the first UPDATE transaction, the j'th client has no pending operations and so its pending operations become simply τ_i. The server then updates its own copy of the document from s_0 to $\tau_i(s_0)$ and, since the i'th client also has no pending operations, returns nothing. So far this is all much the same as before. The difference now is that the user may have made further changes to the i'th client's editable copy of the document by the time the transaction is completed, therefore the editable copy becomes $\tau_i'(\tau_i(s_0))$. This means that there is no use in comparing the server's copy of the document with the i'th client's editable copy in any proof, rather the comparison has to be

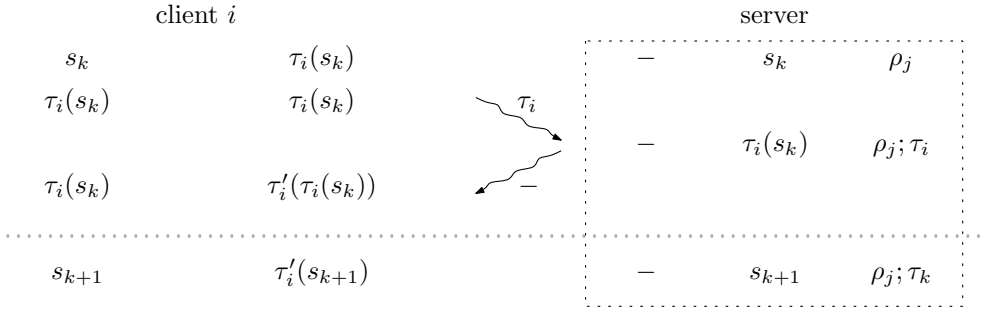


Figure 14: The k'th UPDATE transaction in the general case with no pending operations

with its working copy. In this case it is easy to see that the two remain in line. Finally, we rename τ_i to τ_0 and set $\tau_0(s_0)$ to s_1.

We next consider the k'th UPDATE transaction. There are two possibilities: either the i'th client completed the previous transaction and therefore has no pending operations, or it did not. Figure 14 illustrates the first possibility, with τ_i and τ_i' being re-used. Note that there is an important difference between this transaction and the first, namely that we cannot now assume that the i'th client's working and editable copies start off in line. In the case of the first UPDATE transaction this could be assumed, because the implementation ensures that user interactions are discarded until the INITIALISE transaction has completed, at which point the editable copy is set to be in line with the working copy. Now this is not the case and the difference τ_i between the editable and working copies could have come about either after completion of the previous transaction or whilst it was in progress. Either way it does not matter, however we draw attention to the fact that the topmost line showing the editable and working copies as being in line is missing, to make the point. Also note that should this k'th UPDATE transaction be the second, we would equate the τ_i here with the previous τ_i'.

To continue, this transaction unfolds in a similar way to the first. Because the j'th client now has pending operations ρ_j, the i'th client's operations τ_i are appended to these. We then rename τ_i to τ_k and set $\tau_k(s_k)$ to τ_{k+1}. Again the i'th client's working copy and the server's copy of the document, both being s_{k+1}, end up being in line if we assume that both start as being so. This assumption forms the induction hypotheis in an inductive proof based on the number of UPDATE transactions, of which the first transaction is the base case. We present that proof once the theory is explained.

Now we come to the other possibility for the k'th UPDATE transaction, namely that the i'th client did not initiate the previous transaction. Figure 15 illustrates this. Here the i'th client has pending operations ρ_i and its working copy of the document s_l, for $l < k$, will be equal the server's copy of the document after the $l-1$'th UPDATE transaction, namely the last transaction initiated by the i'th client itself. The following lemma will prove useful:

Lemma 4.2. $\rho_i(s_l) = s_k$

Proof. We just have to observe that ρ_i is $\tau_l; ...\tau_{k-1}$ and since $\tau_l(s_l) = s_{l+1}$ all the way up to $\tau_{k-1}(s_{k-1}) = s_k$, the result follows. □

As usual the operations τ_i the i'th client sends to the server are transformed relative to it's pending operations to become $\tau_i \backslash \rho_i$ before the server applies them to its copy of the document and appends them to the j'th client's pending operations ρ_j. And again, as usual

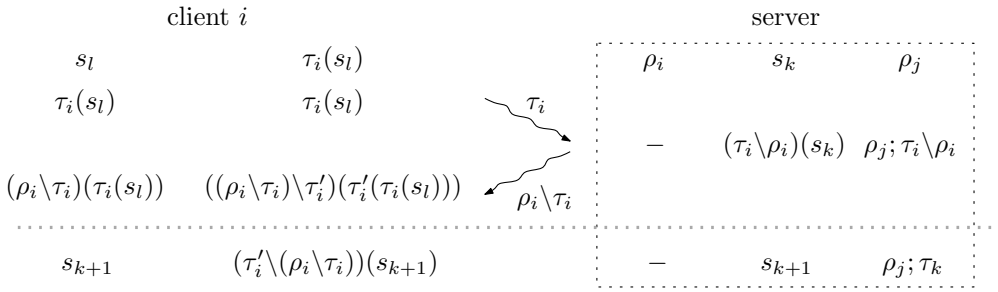


Figure 15: The k'th UPDATE transaction in the general case with pending operations

rather than return the i'th client's pending operations directly, the server first transforms them relative to the operations τ_i to become $\rho_i \backslash \tau_i$. The i'th client duly applies them to its working copy of the document s_l to give $(\rho_i \backslash \tau_i)(\tau_i(s_l))$. Now we rename not τ_i but $\tau_i \backslash \rho_i$ to τ_k and set $\tau_k(s_k)$ to τ_{k+1}. In order to show that the i'th client's working copy and the server's copy of the document remain in line we have the following lemma:

Lemma 4.3. $(\rho_i \backslash \tau_i)(\tau_i(s_l)) = s_{k+1}$

Proof.

$$\begin{aligned}
(\rho_i \backslash \tau_i)(\tau_i(s_l)) &= (\tau_i ; \rho_i \backslash \tau_i)(s_l) \\
&= (\rho_i ; \tau_i \backslash \rho_i)(s_l) \\
&= (\tau_i \backslash \rho_i)(\rho_i(s_l)) \\
&= (\tau_i \backslash \rho_i)(s_k) \\
&= \tau_k(s_k) \\
&= s_{k+1}
\end{aligned}$$

Here we have made use of lemma 4.2 and the usual identities. □

Lastly the i'th client must also apply its transformed pending operations $\rho_i \backslash \tau_i$ to its editable copy of the document. Further changes to this copy during the course of the transaction however means that it has become $\tau_i'(\tau_i(s_i))$ and therefore the pending operations must be transformed again, this time relative to τ_i', becoming $(\rho_i \backslash \tau_i) \backslash \tau_i'$, before being applied. The editable copy therefore becomes $((\rho_i \backslash \tau_i) \backslash \tau_i')(\tau_i'(\tau_i(s_l)))$, as illustrated in figure 15. A proof along similar lines to that given in lemma 4.3 will equate this handful to $(\tau_i' \backslash (\rho_i \backslash \tau_i))(s_{k+1})$ but for the sake of the reader we omit this.

We end this section by summarising these results formally:

Theorem 4.1. Consider a fixed number of clients, each having completed an INITIALISE transaction. Then after any subsequent UPDATE transaction, the working copy of the client that completed the transaction and the server's copy of the document are in line.

Proof. By induction on the number of UPDATE transactions. Figure 13 illustrates the base case of the first UPDATE transaction. Figures 14 and 15 illustrate that if, after the k'th UPDATE transaction, the working copy of client that completed the transaction and the server's copy of the document are in line then, after $k+1$'th UPDATE transaction, this is also true. So this holds after any number of UPDATE transactions. □

Finally, we can drop the requirement that the number of clients be fixed:

Theorem 4.2. Consider an increasing number of clients. Then after any transaction, the working copy of the client that completed the transaction and the server's copy of the document are in line.

Proof. Suppose a client completes an INITIALISE transaction. Then its working, and indeed editable, copies are in line with the server's copy of the document when the INITIALISE transaction completes. Moreover since the pending operations of the other clients and the server's copy of the document remain unchanged, after any subsequent UPDATE transaction, the working copy of the client that completed the transaction and the server's copy of the document are in line by theorem 4.1. Should other clients complete an INITIALISE transaction, the same argument can be used. □

Aside from the proof that our operational transformations preserve the intention of each individual operation, found in subsection 5.3, the proofs in this section together with those in sections 2 and 3 give the first ever formally correct concurrency control algorithm for collaborative text editors.

5. Consistency

In this section we prove that our algorithm is correct against the standard consistency model, although as we mentioned in the introduction, we hope that our algorithm's correctness has been shown to be self-evident without recourse to consistency models. Furthermore it is not unreasonable to remark that the standard consistency model can be a little problematic in places. We therefore modify it as we go along before drawing parallels.

Known as the CCI model, the standard consistency model requires the following three properties of a concurrency control algorithm to hold:

- **C**onvergence
- **C**ausality, or precedence preservation
- **I**ntention preservation

We tick these off one by one in what follows.

5.1. **Convergence.**

To begin with we describe the concept of quiescence in the context of concurrency control algorithms. It is a time when there are no operations left to be executed by any client. A concurrency control algorithm is said to be convergent if it ensures that all the client's copies of the document are in line at quiescence, whenever this occurs. Immediately this is problematic, since there is no such thing as global time in a distributed system, as we know. We therefore redefine both quiescence and convergence, rather than relying on something akin to our own previous woolly notion of global time.

We first define what we call local quiescence. Consider the response part of any transaction. We know that the client no longer has pending operations on the server the moment the response is sent, and also that if it contains any pending operations they are immediately executed on the client the moment it is received. Therefore we define local quiescence as a combination of the moment on the server that the response is sent together with moment it happens before, namely the moment on the client when it is received. This allows us to define what we call local convergence as the property that the client's working copy and

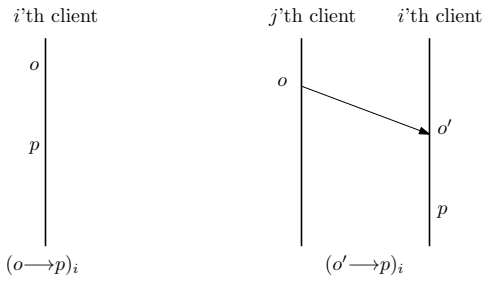


Figure 16: The definition of $o \to p$.

the server's copy of the document should be in line at local quiescence. So to say that our algorithm is locally convergent is no more than a restatement of theorem 4.2.

In choosing to compare the working copies of client's documents with the server's copy rather than their editable copies it may appear as if we are making a compromise. As mentioned in section 4, however, it is impossible to do any better. Recall that the editable copy is considered to be no more than the value of the input field itself, therefore nothing can be said about it being in line with the server's copy at any given moment beyond remarking that if no user interactions take place during the course of a transaction, then the client's working and editable copies of the document will be in line when the transaction completes. Algorithms that execute operations the moment they are generated, as ours does, are called optimistic [CNDL95].

Is there a more general proof that is closer in spirit to the standard consistency model's definition of convergence? The only moments in time that we have to work with, so to speak, are those moments on the server immediately a response is sent. If we take the pending operations the server has stored for every other client at any such moment and apply them to the working copies of these clients at the moments they completed their last transaction, it turns out not unsurprisingly that resulting copies are all in line. However we think that this is irrelevant, and do not give the proof. What we think is relevant is that over a series of transactions the working copy of the client that completes any transaction and the server's copy of the document are always in line at the moment the transaction is completed, and so we leave it at that.

5.2. **Causality or precedence preservation.** So far we have not mentioned causality, or precedence as it is also known, because it played no part in the development of our algorithm. Preserving causality is an issue for algorithms based on a peer-to-peer model, where the lack of a centralised controlling process makes keeping track of the order of operations extremely difficult as they are bandied about in all directions. By contrast we could say that our algorithm preserves causality a priori, since it has never been an issue. Nonetheless we give the standard definition of causality, which is based on Lamport's "happens before" relation, together with a proof that our algorithm preserves it.

To begin with we couch the generation and execution of operations on clients formally in terms of events. Operations are written o and p, their transformed counterparts o' and p' respectively. We write o_i to represent the event of operation o being generated on the i'th client and p'_j, say, to represent the event of the transformed operation p' being executed by the j'th client. We write $o_i \to p_i$ if o_i occurs before p_i, $o'_i \to p_i$ if o'_i occurs before p_i and so on. Now we can give a definition of causality in terms of events:

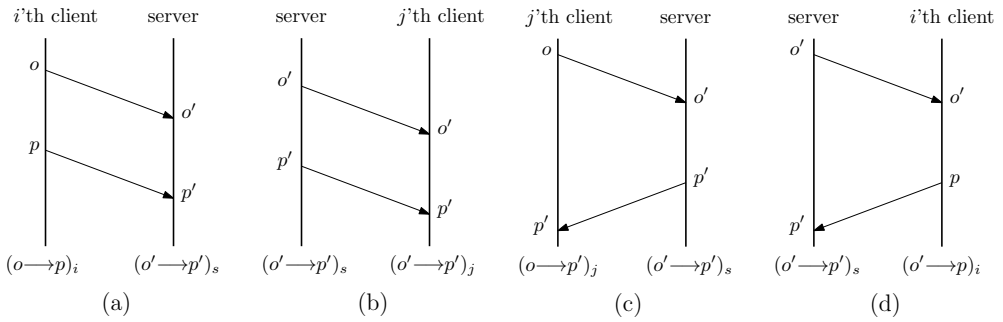


Figure 17: Operations are always communicated in sequential order.

Definition 5.1. The operation o causes, or precedes, the operation p, written $o \to p$, when either $o_i \to p_i$ or $o'_i \to p_i$.

We abbreviate $o_i \to p_i$ as $(o \to p)_i$ and $o'_i \to p_i$ as $(o' \to p)_i$. Also $o'_i \to p'_i$ can happen, which we abbreviate $(o' \to p')_i$. So $o \to p$ when either $(o \to p)_i$ or $(o' \to p)_i$. For an illustration that should make things clearer see figure 16. It should also be clear that if $o \to p$ we cannot have $p \to o$. This very valuable contribution, namely the realisation that Lamport's "happens before" relation on events leads to a relation on operations, is due to [EG89].

Now we come to the definition of the preservation of causality. An algorithm is said to preserve causality if, whenever $o \to p$, o is executed before p on all clients. Formally:

Definition 5.2. An algorithm preserves causality when the following implications hold:

$$(o \to p)_i \Rightarrow (o' \to p')_j$$
$$(o' \to p)_i \Rightarrow (o' \to p')_j \wedge \exists k\, (o \to p')_k$$

Here we assume as usual that $j \neq i$, and also that $i, j \neq k$.

Now we formalise the notion of operations being put on the server. It is somewhat arbitrary whether we write $(o' \to p')_s$ or $(o \to p)_s$ here. We choose the former to emphasise the fact that operations are transformed the moment they arrive.

Definition 5.3. If o is put on the server before p we write $(o' \to p')_s$.

We next formalise the assumptions made in section 4 relating to the sequential order of operations on both client and server. Again for an illustration should make things clearer, see figure 17.

Assumption 5.1. If $(o \to p)_i$ then $(o' \to p')_s$.

Assumption 5.2. If $(o' \to p')_s$ then $(o' \to p')_j$.

Assumption 5.3. If $(o \to p')_i$ then $(o' \to p')_s$ and vice versa.

Assumption 5.4. If $(o' \to p')_s$ then $(o' \to p)_j$ and vice versa.

The proof that our algorithm preserves causality is now straightforward.

Lemma 5.1. Our algorithm preserves causality.

Proof. We make use of figure 17. If $(o \to p)_i$ we join parts (a) and (b) to get $(o' \to p')_j$. If $(o' \to p)_i$ then by part (d) we must have $(o' \to p')_s$. Then by part (b) we have $(o' \to p')_j$ except for one client, since o must be generated somewhere. Then by part (c) we have $(o \to p')_j$ for one value of j, say k, that is $\exists k\, (o \to p')_k$ and we are done. □

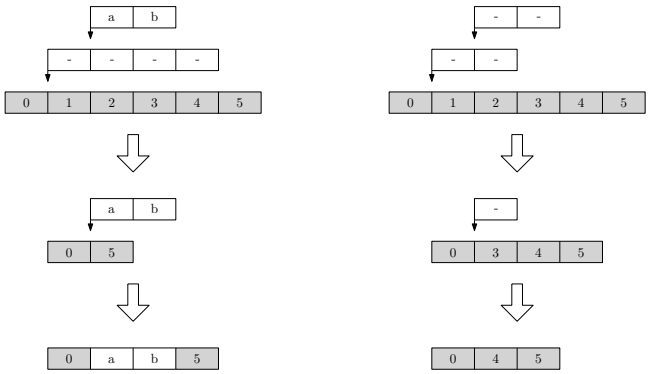


Figure 18: Subtleties when transforming operations relative to deletes.

5.3. **Intention preservation.** This property relates not to the workings of concurrency control algorithms as a whole but only to operational transformations. It turns out that precisely what it means for the intention of an operation to be preserved under transformation is not quite straightforward to formalise. However we jump through the hoops and prove that our stringwise operational transformations preserve intention in a way that hopefully appeals to common sense.

Roughly speaking, the intention of an insert is preserved if, once transformed, it still inserts the same characters in the same place. There is one caveat, illustrated in figure 18 (a), namely that the characters cannot necessarily be inserted in the same place if the left hand corner of the insert overlaps the transforming delete. In this case the transformed insert does the best that it can, so to speak, inserting its characters immediately to the right of the deleted characters. Nonetheless it is reasonable to state that the intention of the insert is preserved. If we consider the string after both operations have been executed, we find that the inserted characters are where we would expect them to be.

In a similar vein, roughly speaking the intention of a delete is preserved if, once transformed, it still deletes the same characters. Again there is a caveat, illustrated in figure 18 (b), namely that the characters cannot all be deleted if the transforming delete has already deleted some or all of them. This is easily accounted for by simply insisting that the transformed operation deletes only those characters that remain. Nonetheless again it is reasonable to state that the intention of the first delete is preserved, even though the second delete did part of the job for it, so to speak. If we consider the string after both operations have been executed, we find that the only requisite characters have been deleted.

We claim that our stringwise operational transformations always preserve the intention of operations. We have just given two of the subtler cases and, admittedly, there are others. Recall from section 2, for example, the transformation of a delete by an insert which results in the delete being split in two. It is easy to check that the intention of the delete is preserved in this case, however. It does, after all, delete the same characters after the transformation as it would have done before. This is illustrated in figure 1.

We could leave it at that, however a proof is needed. So we begin with the formalising the intent of operations and hope that it clarifies rather than occludes:

Definition 5.4. The intention of an insert is an ordered pair consisting of the index of the character immediately to the left of which the insert operation's characters are to be inserted or zero if there are no characters, together with a string of the insert operation's characters themselves. The intention of a delete is a set of the indexes of the characters it deletes:

$$[\![i(n,s)]\!] = (n,s)$$
$$[\![d(n,l)]\!] = \{n, ...n+l-1\}$$

Now we employ a little sleight of hand in order to ease the formalism that follows. Looking at figure 18 (a), we see that the character with index 5 keeps this index after the execution of the first operation, and again, after the second operation. The characters in figure 18 (b) also keep their original indexes in this way. In fact this has always been the case in these illustrations, see figures 1, 2 and 3 in section 2, for example.

When the transforming operation is an insert this renumbering makes the preservation of intention easy to formalise. For the transformation of an insert $i(n,s)$ relative to another insert $i(n',s')$, it should be clear that if we allow characters to keep their original indexes we have:

$$[\![i(n,s) \backslash i(n',s')]\!] = (n,s)$$

Similarly for the transformation of a delete (n,l) relative to an insert $i(n',s')$, again it should be clear that if we allow characters to keep their original indexes we have:

$$[\![d(n,l) \backslash i(n',s')]\!] = \{n, ...n+l-1\}$$

And so in all cases when τ is an arbitrary operation and ρ is an insert we have:

$$[\![\tau \backslash \rho]\!] = [\![\tau]\!] \tag{5.1}$$

When the transforming operation is a delete we know that the cases can be more subtle. For the transformation of an insert $i(n,s)$ relative to a delete $d(n',l')$ we have:

$$[\![i(n,s) \backslash d(n',l')]\!] = \begin{cases} (n'+l', s) & n' \leqslant n < n'+l' \\ (n,s) & \text{otherwise} \end{cases}$$

In other words, if the left hand corner of the insert overlaps the delete, the insert is effectively moved immediately to the delete operation's right. See the illustration on left hand side of figure 1 again. With a little care we can re-use the formalism of section 2:

$$[\![\tau \backslash \rho]\!] = \begin{cases} [\![\tau \uparrow \rho^+]\!] & \tau \simeq \rho \vee \tau > \rho \\ [\![\tau]\!] & \text{otherwise} \end{cases} \tag{5.2}$$

For the transformation of a delete $d(n,l)$ relative to another delete $d(n',l')$ we can do better. If the deletes do not overlap, the transformed delete will delete the same characters, otherwise it will only delete the characters left by the transforming delete. Formally:

$$[\![d(n,l) \backslash d(n',l')]\!] = \{n, ..n+l-1\} \backslash \{n', ..n'+l'-1\}$$

So in the case of both τ and ρ being deletes we have:

$$[\![\tau \backslash \rho]\!] = [\![\tau]\!] \backslash [\![\rho]\!] \tag{5.3}$$

And so we have a proof of sorts.

Lemma 5.2. Our stringwise operational transformations preserve intention.

Proof. It is straightforward to check that in all cases that equalities 5.1, 5.2 and 5.3 hold. □

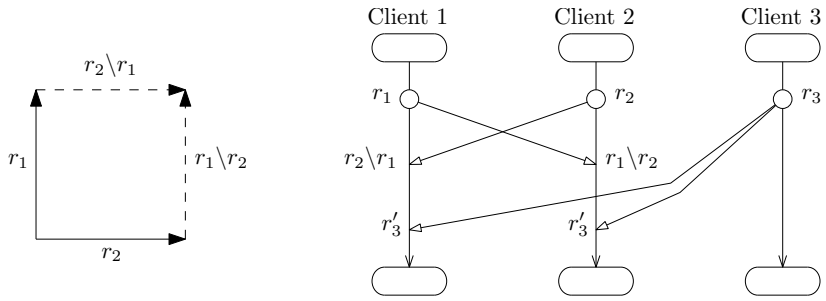


Figure 19: Transformation properties TP1 and TP2

6. Related work and conclusions

To the best of our knowledge ours is the first formally correct concurrency control algorithm for collaborative text editors and it is reasonable to ask why, given the considerable amount of literature surrounding this problem. See [otf] for a partial synopsis, for example, and [LL10] for the following negative view: "proofs are very complicated and error-prone...we can only conclude that an algorithm achieves convergence but cannot draw any conclusion about intention preservation."

Our work contradicts this view. For example we appear to be the first in proving that our operational transformations preserve intention, with more than one recent work [LL05, WUM08] agreeing with our own findings that this has never been done before. In doing so, however, we run the risk of occluding with formalism what is intuitively pretty obvious. When proving convergence on the other hand the formalism feels a lot less strained, in fact we think that the theorems in sections 2, 3 and 4 are genuinely useful in demonstrating that our algorithm works perfectly in practice. We also hope that the crux of the argument, illustrated in figure 12, is perfectly clear. The rest, as they say, then follows.

It is perhaps a little surprising then that stringwise operational transformations such as ours have never been adopted. In our opinion the reason is that they lead to a convergence problem requiring a novel proof as opposed to a simple inductive one. As mentioned earlier in section 2, less than ideal stringwise operational transformations have been used in the past in order to admit inductive convergence proofs, but as a consequence they do not preserve intention [Cor95]. Stringwise operational transformations have occasionally cropped up elsewhere [SYZC96] but the details are vague. Another work [SJZ+98] does indeed acknowledge the effects of what we call fragmentation, but there are no proofs and apparently the implementation led to "complications". Lastly a more recent work [SLG09] uses stringwise operations but splitting deletes is avoided, apparently in order to "simplify presentation and stay focused on the main contribution".

On the other hand, in spite of their limitations in our view, characterwise operational transformations have always been used. Despite the fact that they mitigate against fragmentation, in our opinion making any consistency proof considerably easier, nonetheless a convincing proof against the standard consistency model for an algorithm that uses them seems never to have emerged. Any such proof must start with the result that the operational transformations themselves are correct, and such proofs are lacking from the earliest attempts [EG89, RNRG96, SJZ+98, SCF98], indeed the operational transformations used in these attempts have all been found to be incorrect in [IOR03].

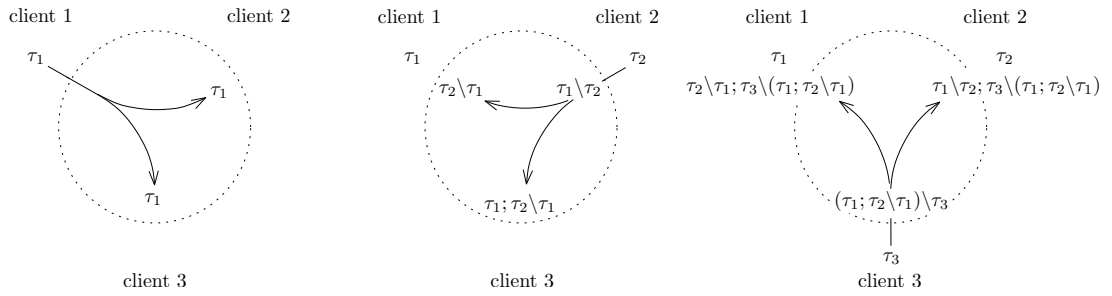


Figure 20: Imposing a total ordering on concurrent operations

This work also contains what appears to be the first set of characterwise operational transformations that have been proved correct using an automatic theorem prover [spi]. By way of comparison we attenuate our own stringwise operational transformations to characterwise ones:

$$i(n,c)\backslash i(m,d) = \begin{cases} i(n,c) & n < m \\ i(n,c) & n = m \wedge c < d \\ i(n,c) & n = m \wedge c = d \\ i(n+1,c) & n = m \wedge c > d \\ i(n+1,c) & n > m \end{cases} \quad i(n,\_)\backslash d(m) = \begin{cases} i(n,\_) & n \leqslant m \\ i(n-1,\_) & n > m \end{cases}$$

$$d(n)\backslash i(m,\_) = \begin{cases} d(n) & n < m \\ d(n+1) & n \geqslant m \end{cases} \quad d(n)\backslash d(m) = \begin{cases} d(n) & n < m \\ e() & n = m \\ d(n-1) & n > m \end{cases}$$

Aside from two small differences these agree with those given in [IOR03]. The first of these is that their algorithm tries to differentiate between inserts with the same position by keeping track of their original position before resorting to a lexicographical ordering on the characters. This is perhaps a little over-engineered. The second is that when both inserts are identical, their algorithm transforms one of the two inserts into the empty operation. We think this is a move towards transformations being in some way bound to the meaning of the underlying content, and cannot agree with it. However these are small gripes.

Moving on, with the problem of finding a correct set of characterwise operational transformations apparently solved, why then did a convergence proof still remain elusive? In our opinion one of the main reasons is a predilection for the peer-to-peer model that has continued from the early days [EG89, Cor95] into recent times [LL07, WUM10] whereas implementations based on the client-server model have always been rare [CNDL95]. The peer-to-peer model brings with it the disadvantage of the lack of a centralised controlling process and, as a consequence, the problem of imposing a total ordering on the operations for the purposes of transformation [VCFS00, LL07], or making do without one. With operations executed concurrently any imposed total ordering is bound to be somewhat arbitrary, however without one the problem becomes harder. In these cases the operational transformations have to satisfy not only our own equivalence 2.1, which has always been known as transformation property TP1 [RNRG96], but also the mysterious transformation property TP2 [RNRG96]. We reproduce the commonplace illustrations of these properties in figure 19, including the first because of its resemblance to our own decreasing diagrams.

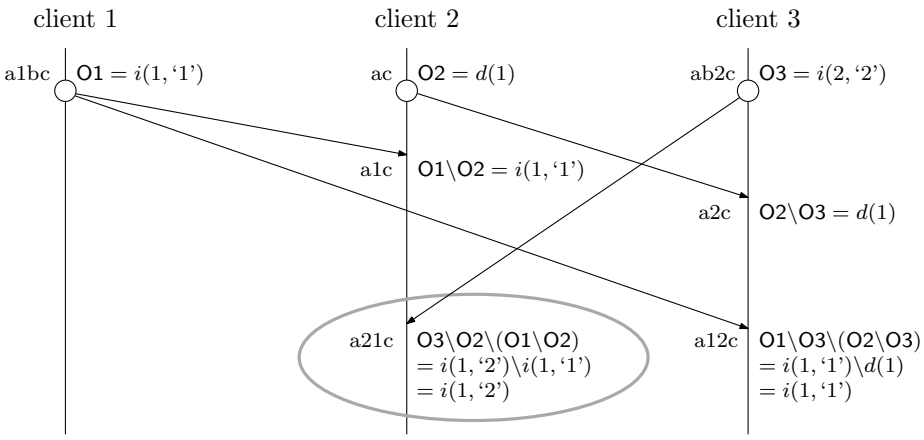


Figure 21: The amended false-tie puzzle

We give an explanation of this mysterious TP2. Whilst TP1 relates to the effect of operations on the underlying document, TP2 requires that operational transformations must ensure that the combined effect of two operations, not on the underlying document but on a third concurrent operation, must be the same regardless of their order, given of course that the second is transformed relative to the first:

$$\tau_3\backslash(\tau_1;\tau_2\backslash\tau_1) = \tau_3\backslash(\tau_2;\tau_1\backslash\tau_2)$$

Incidentally, whether our own operational transformations satisfy this property is moot. Because our algorithm is based on a client-server model rather than a peer-to-peer one, it is able to impose a total ordering on operations for the purposes of transformation by virtue of the fact that transactions are handled sequentially. We show a simple case of how this total ordering is imposed in figure 20, where $\tau_1 < \tau_2 < \tau_3$ on the server. Here pending operations are shown inside the dotted circles whilst operations already executed on the clients are shown outside. Note that the ordering $\tau_1 < \tau_2$ does not mean that τ_2 is never executed before τ_1 on some clients, transformed or otherwise. At client 2, for example, τ_2 will be executed before $\tau_1\backslash\tau_2$. What it does mean, however, is that on the server τ_2 never occurs before τ_1, transformed or otherwise, and since transformations always happen on the server, when any other operation is transformed relative to these two operations, it is always transformed relative to τ_1 before τ_2, again transformed or otherwise. Thus we see $\tau_3\backslash(\tau_1;\tau_2\backslash\tau_1)$, but would never see $\tau_3\backslash(\tau_2;\tau_1\backslash\tau_2)$.

The false tie puzzle. We called the transformation property TP2 mysterious and we give the reasons why. We look at what is called the false tie puzzle [SZJY97], which could be said to test the correctness of operational transformations when a total ordering on operations for the purposes of transformation has not been imposed. We reproduce the commonplace illustration of this puzzle in figure 21, missing out some of the operations executed at the first client because they are not in fact needed for the complete puzzle.

The "puzzle" is that the algorithm diverges because of what is known as a false tie, a seemingly incorrect operational transformation that results from two inserts occupying the same position:

$$i(1, \text{'2'})/i(1, \text{'1'}) = i(1, \text{'2'})$$

Note that this operational transformation is different from our own, as it leaves the lexicographically greater of the two inserts in place rather than the lexicographically lesser when the two inserts are tied. In fact our operational transformation would not break the puzzle but this is not the point. The operational transformation above is just as valid, with the choice of whether to leave the lexicographically lesser or greater insert in place when two inserts are tied being an arbitrary one. The point is that this puzzle cannot differentiate between correct and incorrect sets of operational transformations. What is wrong is not the operational transformations themselves but the algorithm implicit in the puzzle itself.

It is perhaps not surprising that one solution to this puzzle is to require that the operational transformations satisfy TP2. Comparing the sequences of operations executed by the second and third clients, rewriting on occasion and employing TP1 where necessary we get:

$$\begin{aligned}
O2; O1\backslash O2; O3\backslash O2\backslash(O1\backslash O2) &\equiv O3; O2\backslash O3; O1\backslash O3\backslash(O2\backslash O3)\\
... &\equiv O2; O3\backslash O2; O1\backslash O3\backslash(O2\backslash O3)\\
O1\backslash O2; O3\backslash O2\backslash(O1\backslash O2) &\equiv O3\backslash O2; O1\backslash O3\backslash(O2\backslash O3)\\
O1\backslash O2; (O3\backslash O2)\backslash(O1\backslash O2) &\equiv ...\\
O3\backslash O2; (O1\backslash O2)\backslash(O3\backslash O2) &\equiv O3\backslash O2; O1\backslash O3\backslash(O2\backslash O3)\\
(O1\backslash O2)\backslash(O3\backslash O2) &\equiv O1\backslash O3\backslash(O2\backslash O3)\\
O1\backslash O2\backslash(O3\backslash O2) &\equiv ...\\
O1\backslash(O2; O3\backslash O2) &\equiv O1\backslash(O3; O2\backslash O3)
\end{aligned}$$

This begs the question, are there any operational transformations that satisfy TP2? Certainly none of those studied in [IROM06], only the characterwise ones outlined in [IOR03], the ones closely in agreement with our own attenuated stringwise operational transformations. We also agree with [IROM06] that it is a difficult if not impossible task to verify whether a set of operational transformations satisfies TP2 without the help of an automated theorem prover. There are simply too many cases to consider. We wonder at this point whether our own stringwise operational transformations do so but then, as we have pointed out, the matter is moot.

Group undo. Another mystery is the apparent problem of undoing operations in a collaborative environment, so-called group undo. Like other issues this has continued from the early days into recent times [PK94, RNRG96, Sun02, WUM08, SS09]. [Sun02] says that it is "technically challenging and none of the existing group undo solutions is able to offer such a capability", [WUM08] says that "consistency of shared data with the undo feature is a complex issue", and so on.

In all honesty this never occurred to the author. Whether an operation comes about as a result of user interaction or whether programmatically is entirely irrelevant to its inclusion in the client's undo buffer on modern systems. Furthermore when the user presses ctrl-z, say, the last operation in the undo buffer is undone, effectively resulting in a new operation being generated which is indistinguishable from one generated by user interaction or programmatically. The point is the system's undo buffer has nothing to do with our algorithm, and operations generated by way of undos have no special place in it. It simply makes no difference how an operation is generated and there is certainly no need to associate an operation that comes about as a result of an undo with the operation that it undoes. We leave it there.

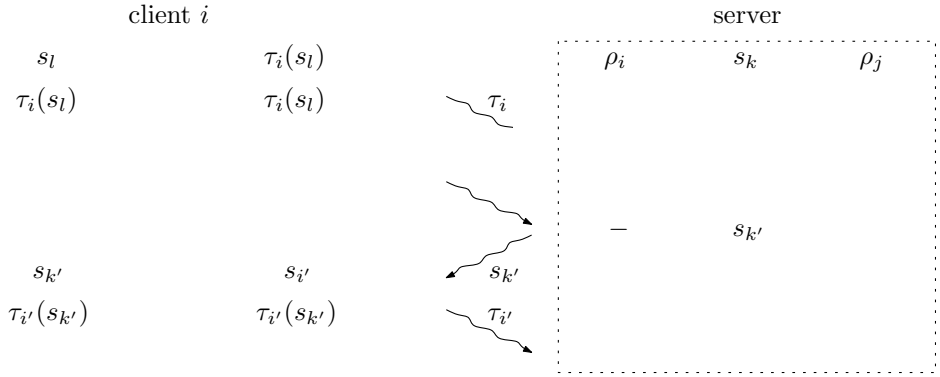


Figure 22: The i'th client re-initialises to become the i''th client

We end on a positive note. Our algorithm is perfectly fault tolerant, by which we mean that the failure of any particular UPDATE transaction to complete does not adversely affect a session. Figure 22 illustrates this. Here the k'th UPDATE transaction fails, and so after a timeout the i'th client starts another INTIALISE transaction, effectively becoming a new i''th client with editable copy of the document remaining intact. The moment the INTIALISE transaction completes new operations are generated by comparing the working and editable copies, thus $\tau_{i'}(s_{k'}) = s_{i'}$ where $s_{k'}$ is the new working copy returned by the server and $s_{i'}$ is the editable copy, uninterrupted so to speak. Immediately after this a new UPDATE transaction can be started. Note that in figure 22 it is the request that fails but it makes no difference to the argument if it is the response that fails. In this case the server's state would be updated but otherwise the situation remains the same. Note also that this whole process can be undertaken unbeknownst to the user.

References

[CNDL95] David Curtis, Pavel Nichols, Michael Dixon, and John Lamping. High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 111–120. ACM, 1995.

[Cor95] Gordon Cormack. A Calculus for Concurrent Update. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 269–279. ACM, 1995.

[EG89] Ellis and Gibbs. Concurrency Control in Groupware Systems. *SIGMOD*, 18(2):399–407, 1989.

[Eng68] Doug's Demo `http://www.dougengelbart.org/firsts/dougs-1968-demo.html`, 1968.

[IOR03] Abdessamad Imine, Pascal Moland Gérald Oster, and Michaël Rusinowitch. Proving Correctness of Transformation Functions in Real-time Groupware. In *European Conference on Computer-Supported Cooperative Work*, pages 277–293. Springer, 2003.

[IROM06] Abdessamad Imine, Michal Rusinowitch, Grald Oster, and Pascal Molli. Formal Design and Verification of Operational Transformation Algorithms for Copies Convergence. *Theoretical Computer Science: Algebraic Methodology of Software Technology*, 351(2):167–183, 2006.

[KKP] Sanjeev Khanna, Keshav Kunal, and Benjamin Pierce. A Formal Investigation of `Diff3` `http://www.cis.upenn.edu/~bcpierce/papers/diff3-short.pdf`.

[KvOdV00] Jan Willem Klop, Vincent van Oostrom, and Roel de Vrijer. A Geometric Proof of Confluence by Decreasing Diagrams. *Journal of Logic In Computing*, 10(3):437–460, 2000.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[LL05] Rui Li and Du Li. Commutativity-based Concurrency Control in Groupware. In *1st International Conference on Collaborative Computing*, pages 10–pp. IEEE, 2005.

[LL07] Rui Li and Du Li. A New Operational Transformation Framework for Real-time Group Editors. *IEEE Transactions on Parallel Distributed Systems*, 18(3):307–319, 2007.

[LL10] Rui Li and Du Li. An Admissibility Based Operational Transformation Framework for Collaborative Editing Systems. *Computer Supported Cooperative Work*, 19(1):1–43, 2010.

[LPS09] Mihai Letia, Nuno Preguiça, and Marc Shapiro. CRDTs: Consistency without Concurrency Control. Technical report, INRIA, 2009.

[New42] Max Newman. On Theories with a Combinatorial Definition of Equivalence. *Annals of Mathematics*, 43(2):223–243, 1942.

[otf] Operational Transformation FAQ `http://www3.ntu.edu.sg/home/czsun/projects/otfaq`.

[OUI05] Gérald Oster, Pascal Urso, and Pascal Moland Abdessamad Imine. Real-time Group Editors without Operational Transformation. Technical report, INRIA, 2005.

[PK94] Atul Prakash and Michael Knister. A Framework for Undoing Actions in Collaborative Systems. *ACM Transactions on Computer-Human Interaction*, 1(4):295–330, 1994.

[RNRG96] 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*, pages 288–297. ACM, 1996.

[SCF98] Maher Suleiman, Michèle Cart, and Jean Ferrié. Concurrent Operations in a Distributed and Mobile Collaborative Environment. In *Proceedings of the 14th International Conference on Data Engineering*, pages 36–45. IEEE, 1998.

[SJZ+98] Chengzheng Sun, Xiaohua Jia, YanChun Zhang, Yun Yang, and David Chen. Achieving Convergence, Causality-preservation, and Intention-preservation in Real-time Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, 1998.

[SLG09] Bin Shao, Du Li, and Ning Gu. An Optimized String Transformation Algorithm for Real-time Group Editors. *2013 International Conference on Parallel and Distributed Systems*, pages 376–383, 2009.

[spi] The SPIKE Automated Theorem Prover `https://github.com/sorinica/spike-prover`.

[SS09] David Sun and Chengzheng Sun. Context-based Operational Transformation in Distributed Collaborative Editing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(10):1454–1470, 2009.

[Sun02] Chengzheng Sun. Undo As Concurrent Inverse in Group Editors. *ACM Transactions on Computer-Human Interaction*, 9(4):309–361, 2002.

[SYZC96] Chengzheng Sun, Yun Yang, Yanchun Zhang, and David Chen. A consistency model and supporting schemes for real-time cooperative editing systems. *Australian Computer Science Communications*, 18:582–591, 1996.

[SZJY97] Chengzheng Sun, Yanchun Zhang, Xiahua Jia, and Yun Yang. A Generic Operation Transformation Scheme for Consistency Maintenance in Real-time Cooperative Editing Systems. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work: the integration challenge*, pages 425–434. ACM, 1997.

[VCFS00] Nicolas Vidot, Michelle Cart, Jean Ferrié, and Maher Suleiman. Copies Convergence in a Distributed Real-time Collaborative Environment. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 171–180. ACM, 2000.

[WUM08] Stéphane Weiss, Pascal Urso, and Pascal Molli. A Flexible Undo Framework for Collaborative Editing. Technical report, INRIA, 2008.

[WUM10] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1162–1174, 2010.