ABSTRACT
Real-time group editing has been envisioned as an effective manner of collaboration. For years, operational transformation (OT) has been the standard concurrency control mechanism for real-time group editing, due to its potential for high responsiveness to local editing operations. OT algorithms are generally non-trivial to be error-free and are computation intensive. Recently, commutative replicated data types (CRDT) have appeared as an alternative to OT. The state-of-the-art OT and CRDT work still lacks the basic functionality found in single-user text editors. In particular, there is no published work that supports both string-wise operations and selective undo. This paper presents an approach that combines and extends OT and CRDT strengths. It is fully decentralized and supports string-wise editing operations and selective undo. Our performance study shows that it provides sufficient responsiveness to the end-users.

Categories and Subject Descriptors
C.2.4 [Computer-Communication Networks]: Distributed Systems—Distributed applications; H.5.3 [Information Interfaces and Presentation]: Group and Organization Interfaces—Collaborative computing

General Terms
Algorithms, Performance

Keywords
Real-time collaborative editor, commutative replicated data type, operation transformation.

1. INTRODUCTION
A real-time group editor allows multiple users to simultaneously edit the same document from different places. Fully decentralized, or peer-to-peer, collaboration has generally the advantage of availability, scalability and resistance to censorship and surveillance, over collaboration via a central server.

Operational transformation (OT) has been established as a concurrency control mechanism for real-time group editing due to its potential for high responsiveness to local operations [4, 5, 6, 7, 8, 10, 14, 15, 16, 17, 18, 19, 21]. Local operations are executed immediately at local peers and later transformed and integrated at remote peers. OT algorithms are sophisticated. Counterexamples of several published OT algorithms have been reported. Moreover, they have time complexity in the length of operation history of the document been edited, which potentially grows endlessly.

Recently, a new class of mechanisms called commutative replicated data types (CRDT) have been proposed [2, 11, 12, 13, 22, 23]. Concurrent operations of a CRDT are mutually commutative, so that a document is eventually kept consistent at all peers.

A real-time group editor should support at least the most basic functionality found in a single-user text editor. At its minimum, it should support insertion and deletion of single characters and strings of characters, as well as the undo and redo of the insertions and deletions. String-wise operations are important as they are the basis for other useful operations like copy-paste, select-delete and find-replace. Surprisingly, there is currently no published work that supports both string-wise operations and their undo.

Our work supports both string-wise operations and their undo by combining and extending existing OT and CRDT approaches.

The rest of this paper is organized as follows. Section 2 presents background and related work. Section 3 gives an overview of the approach. Section 4 presents the view-model architecture and the data structure of the model. Section 5 describes operations and updates in view and model. Section 6 describes how model and view are synchronized. Sections 7 and 8 describe how local and remote operations are integrated into the model. Section 9 shows the correctness of the approach. Section 10 presents performance results. Section 11 discusses some open issues. Section 12 concludes.

2. BACKGROUND
OT was first introduced in [4]. The basic idea is as the following. A shared document is replicated at different peers. An editing operation is first executed at a local peer and then propagated to remote peers. Suppose two peers start with “012”. Peer 1 inserts “a” between “0” and “1”, with ins(1, “a”) and Peer 2 deletes “2” with del(2). The states after local executions at the two peers are “0a12” and “01”. Now if the two peers execute the remote operations as is, the states at these peers become “0a2” and “0a1”, which are inconsistent. With OT, the remote operations are transformed to include the executed concurrent operations, into ins(1, “a”) and del(3) respectively. The two peers are in consistent state “0a1” after executing the transformed operations.

There are some challenges with this basic approach. First, a remote operation can only be transformed to include a concurrent
operation that is compatible, i.e., the two operations operated on exactly the same state. To achieve this, a peer has to first transpose the history of operations to make the operations compatible, and then include the effects of compatible operations. The transposition involves the transformation of both remote operation and operations in the local history. This whole process is sometimes called operation integration. The complexity of operation transformation and integration algorithms, or OT algorithms for short, depends on factors like how operations in the operation history are ordered and whether the integration of remote operations has to follow some restricted order. OT algorithms have linear or quadratic time complexities in the length of the operation history.

Transformation functions are difficult to be made correct. Counterexamples were found for many of the published transformation functions. For instance, [5], [6], [7], [10] and [17] reported counterexamples of earlier work. [8], [10], [11], [14], [15], [16], [17] and [21] are among the few that have no counterexample reported. One source of complication is the role of deleted characters, called landmarks in [6] and tombstones in [10]. Given three peers all starting with “012”. Peers 1, 2, and 3 issue concurrent operations \( \text{ins}(2, 'x'), \text{del}(1) \) and \( \text{ins}(1, 'a') \) respectively. Because ‘x’ is inserted to the right of “1” and “a” to the left of “1”, the final states at all peers must be “0ax2”. The deleted character “1” determines the ordering of “a” and ‘‘x”. However, because “1” is deleted at Peer 2, the two inserted characters thus tie at Peer 2. Most counterexamples are due to failure to break the tie of this type in different combinations of concurrent operations. [20] concludes that this is the only source of puzzles, as far as normal (i.e. not undo) insertion and deletion of single characters are concerned.

Another source of complication is the undo of executed operations. Single-user editors typically support undo of operations in chronological order. In a collaborative editor, operations are not totally ordered chronologically due to concurrent operations. It is therefore necessary to support selective undo to undo the effect of any selected operation that has been executed. An undo operation \( \text{undo(op)} \) is heavily dependent not only on the operation \( op \) it undoes, but also on the contexts in which \( op \) and \( \text{undo(op)} \) are executed. Simply issuing a separate reverse operation \( \overline{op} \) and transforming it as a normal operation ignore such dependencies and therefore may lead to undesirable results in certain situations, such as in the dOPT puzzle [18]. Some particular solutions are introduced to address this issue. For example, in ABTU [15], \( \text{undo(op)} \) and \( op \) are associated with special attributes and are placed next to each other in the operation history. [21] concludes that operation transformation rules based solely on operation causal relations are no longer sufficient to cope with the complexity of existing OT systems, and introduces a fundamentally different context-based OT framework where the dependency of \( \overline{op} \) on \( op \) becomes more explicit in terms of operation contexts.

Due to the inherent sophistication of OT, most published work only supports single-character insertions and deletions. GOT [19] and ABTS [14] are the only algorithms accessible in the literature that support string-wise operations. [7] identified a counterexample of GOT. Moreover, GOT seems to be superseded by the follow-up work of the same researchers in [21]. There has been no published work following [21] that supports string-wise operations. In ABTS [14], an operation history is composed of a sub-history of insersions followed with a sub-history of deletions. A string-wise deletion may be split into a set of sub-deletions during subsequent operation transformations.

Supporting selective undo or string-wise operations alone is already challenging. Supporting both can be harder. For example, ABTU [15] and ABTS [14], of the same researchers, use different (incompatible) operation history structures to support selective undo and string-wise operations. It is not obvious that ABTU and ABTS can be combined to support both selective undo and string-wise operations.

Recently, CRDT has appeared as an alternative to OT for decentralized real-time group editing. With CRDT, concurrent insertions are ordered based on the underlying data structure rather than on operation transformation, so that the time complexity may not depend on the lengths of operation histories. [2, 12, 13, 22, 23] achieve this by making use of specially designed identifiers associated with edited objects (characters, lines of characters) or operations. An identifier contains information about the relative positioning of objects [2, 12, 13, 22, 23] as well as operation causality [13]. In [2], [12] and [22], the sizes of identifiers can grow unbounded, but experiments show that the sizes stay low even in the most demanding situations. [11] takes a different approach. There, every character is uniquely identified and is associated with the previous and next characters at the time of its original insertion. This information is used for the ordering of the characters at remote peers. Experimental results [1] show that CRDT algorithms outperform OT algorithms by a factor between 25 and 1000.

As the CRDT-based approaches are still quite young, only [2] and [24] support string-wise operations and [22] supports undo. There has been no support for undo of string-wise operations.

We can think of the different approaches from a unified perspective. On the one hand, operation histories in OT approaches can be seen as abstract data types and operations that transpose and manipulate the histories exhibit certain properties similar to commutativity of CRDTs. On the other hand, CRDTs can be regarded as operation histories extended with explicit relations among operations. Based on this unified perspective, our work combines and extends the strengths of the two approaches. The data structure is basically a list of nodes (of sub-strings) that forms a total order, similar to the operation history in ABTU [15] that arranges operations in a total effects-relation order (as formally defined in [7]). In addition, the data structure materializes dependencies among operations. Dependencies among insertions are similar to [11]. To support string-wise operations, additional links connect nodes belonging to the same operations. Additional information is also provided for undo of operations.

Our earlier work on support for string-wise operations was reported in [24]. The initial ideas of this paper was first reported in a short note [25].

3. OVERVIEW OF APPROACH

A document is collaboratively edited by a number of peers at different sites. Every peer consists of a view, a model, a log of operation history and three queues (Figure 1).

A peer concurrently receives local operations from the user and remote updates from other peers. A view is mainly a string of characters. A user at a peer can insert or delete a sub-string at a position in the string, and undo an earlier executed local or remote operation. The user’s operations take immediate effect in the view. Local and remote operations are first stored in queues \( Q_{\text{loc}} \) and \( Q_{\text{rem}} \) and later be integrated in the model. Integrated local operations are first stored in \( Q_{\text{int}} \) and later broadcast to other peers. When the model is rendered, the effects of integrated remote operations are shown in the view. Integrated operations are also stored in the log. A user can select an operation in the log to undo.

The model is primarily a double-linked list of nodes. Following [7] and [14, 15], we call the order in which the nodes are linked \( \text{effects-relation order} \). A node contains a sub-string, together with some additional meta-data and links for the operations on the string.
4. DATA STRUCTURE

A peer has an identifier pid and maintains a number pun (peer update number). pun increments by one for every editing operation originated at the peer. An operation is uniquely identified with (pid, pun) at all peers.

A character string, or simply string, is a sequence of characters. A position in a string, represented as a non-negative integer, is a place either left to the leftmost character, between two adjacent characters, or right to the rightmost character, of the string. For string str, we use str[pos] to denote the character right to position pos in str, or nil if pos is the rightmost position of str. Suppose that every character is uniquely identified. We use posstr[c] to indicate the position left to character c in str. For two adjacent characters c₁ and c₂ in a str and c₁ is left to c₂, we have posstr[c₁] + 1 = posstr[c₂].

A view is a character string currently visible to the user. The leftmost position of view is always 0. curview is the current position in view.

A model consists of a set of nodes. A node is originated with an ins operation and we say that a node belongs to its originating ins operation. A node has an interoperable part and a peer-specific part. The element values of the interoperable part of nodes will be preserved at all peers. The interoperable part of a node v contains the following elements:

- pid, pun, the id of the ins operation which v belongs to.
- str, the character string of v. We also use v.len for v.str.len, the length of v.str.
- offset, the leftmost position of v.str with respect to the string of the ins operation it belongs to. When a node is originated with an insertion, its offset is 0. Splitting the node at position pos leads to two nodes, with offsets 0 and pos respectively.

- l, r, the left and right nodes of v in effects-relation order. In Figure 2, the l and r links are illustrated with solid lines.
- i_l, i_r, the left and right nodes of the same ins operation. In Figure 2, the i_l and i_r links are illustrated with dotted lines.
- dep_l, dep_r, insert dependencies, i.e. the place of the originating insertion, represented with the right end of the left node (pid_l, pun_l, offset_l, len_l) and the left end of the right node (pid_r, pun_r, offset_r).
- dels, a set of del elements related to deletions of v.str.
- undo, the undo of the insertion, or nil if the insertion is not undone.

In a peer, a node can be directly referred to via its reference (i.e. a pointer). So the links l, r etc. refer to nodes with their references.

Node references, however, are meaningless across peer boundaries. Fortunately, a node can be uniquely identified by the id of the ins(str) operation that inserted the string str, together with the offset of the node. In Figure 2, suppose the id of ins(“0123456789”) is i_0 = (1, 1). The node with string “56” can be uniquely identified with (1, 1, 5). We use (pid, pun, offset) as the id of a node. Nodes are hash-indexed with their ids. Therefore given (pid, pun, offset), a node can be obtained in near-constant time. In the worst case, if a peer cannot find a node with id (pid, pun, offset), it can start from (pid, pun, 0) and follow i links to find the node containing position offset (and then make a split there).

As a convention in this paper, offset is specifically used to identify nodes’ left ends. For positions in view or inside nodes, we use pos instead.

Definition 1. Node vᵢ is left to node vⱼ (or equally, vⱼ is right to vᵢ), written as vᵢ < vⱼ or vⱼ > vᵢ, if either (a) vᵢ.r = vⱼ (and...
\( v_i = v_j \), or (b) there exist \( v_1, v_2, \ldots, v_n \), such that \( v_1 \prec v_2 \prec \cdots \prec v_n \prec v_r \).

In this paper, we use properties to describe invariants on data structures that are maintained by the algorithms. We use \( \prec \)-order for the effects-relation order among nodes.

**Property 1.** The nodes in a model form a total \( \prec \)-order.

An insertion consists of the nodes chained with the \( i_t \) and \( i_r \) links. For insertion of string \( str \), if \( v \) is the leftmost node of the insertion, \( v.\text{offset} = 0 \); if \( v \) is the rightmost node of the insertion, \( v.\text{offset} = \text{len} + v.\text{len} \); if \( v \) is neither the leftmost nor the rightmost node, \( v.\text{offset} = \text{len} + v.\text{len} \).

A deletion consists of the nodes containing the \( \text{del} \) elements of the same \( \text{del} \) operation. When \( v.\text{del} \) of a node contains multiple \( \text{del} \) elements, \( v.\text{str} \) has been deleted concurrently by multiple peers.

A \( \text{del} \) element has the following (sub-)elements:

- \( v \), the node whose \( \text{del} \) contains this \( \text{del} \).
- \( i_t, i_r \), the left and right \( \text{del} \) elements of the same \( \text{del} \) operation.
- \( \text{undo} \), undo of the deletion, or nil.

**Property 2.** Let \( v_l \) and \( v_r \) be nodes and \( \text{del}_l \) and \( \text{del}_r \) be \( \text{del} \) elements. (a) \( \prec \)-order: \( v_l.\text{r} = v_r \Leftrightarrow v_l = v_r \); (b) Insertion: \( v.\text{str} = v.\text{str} + v.\text{str} \); (c) Deletion: \( v.\text{del} = v.\text{del} - v.\text{del} \).

An undo element consists of the following (sub-)elements:

- \( v \), the node, the \( \text{del} \) or the undo element this undo element is part of.
- \( \text{undo} \), the undo element if this undo itself is undone, or nil otherwise.
- \( \text{pos} \), a set of the ids of the concurrent undo operations of the same operation.

Please notice the different handling of concurrent \( \text{del} \)s and concurrent undo. Concurrent undo always refer to the same operation to be undone, and thus are regarded as a single undo, whereas concurrent \( \text{del} \)s, albeit with overlapping characters, are different operations. Therefore a node maintains a set of \( \text{del} \) elements (with their own sub-elements), but an undo element only maintains the sub-elements once for a set of concurrent counterparts.

If an undo operation itself is undone, the undo element refers to another undo element. An operation’s undo elements are thus chained to a linked list.

**Definition 2.** An operation is effectively undone if it is undone an odd number of times. A node (for insertion) or an element (for deletion or undo) of an operation is effectively undone if the length of the undo chain of the node or element is an odd number.

**Definition 3.** A node \( v \) is visible, written as \( v.\text{visible} \), if it is not effectively undone and all \( \text{del} \in v.\text{del} \) are effectively undone.

A node \( v \) has the following peer-specific elements:

- \( \text{pos}^r \), a number indicating its relative position in the \( \prec \)-order of all nodes.
- \( \text{render} \), true if the node’s visibility is reflected in the view.

- \( \text{strInView} \), true if the string of the node is currently shown in the view.

**Property 3.** \( v_1 \prec v_r \Leftrightarrow v_1.\text{pos}^r < v_r.\text{pos}^r \).

Note that the \( \text{pos}^r \) values of nodes are not globally unique. When a node is inserted between two nodes, any \( \text{pos}^r \) value between the \( \text{pos}^r \) values of the two nodes can be chosen. If there is no new value available, we can simply re-assign the \( \text{pos}^r \) values among a set of nodes to make sufficient interval between two adjacent values. \( \text{pos}^r \) is a simple implementation of a order-maintenance list [3].

**Property 4.** \( \text{v.rendered} \Rightarrow \text{v.visible} = \text{v.strInView} \).

A node \( v \) can be split at \( \text{pos} \), where \( \text{v.offset} < \text{pos} < \text{v.offset + v.len} \). This results in an updated \( v \), denoted here as \( v_l \), and a new right-hand node \( v_r \), such that

- \( v_l.\text{offset} = \text{pos} \land v_l.\text{len} = \text{pos} - v_l.\text{offset} \land v_l.\text{str} = v_l.\text{str} \).
- All elements of \( v \) are preserved in \( v_l \).
- \( v_l.\text{undo} \) refers to \( v_l.\text{undo} \) and all other elements are deep-copied from \( v_l \) to \( v_r \).
- \( v_r \) is inserted in all linked lists that \( v \) involves to maintain the encapsulated operations, including the \( l-r \) and \( i_t-i_r \) links as well as the \( l-r \) links of \( v_l.\text{del} \) elements.

Property 3 is maintained.

**Property 5.** The left and right ends of node \( v \), represented with \( (v.\text{pid}, v.\text{pun}, v.\text{offset}) \) and \( (v.\text{pid}, v.\text{pun}, v.\text{offset}, v.\text{len}) \), can be uniquely located in a model, despite subsequent splits.

If node \( v \) is later split into \( v_l \) and \( v_r \), \( (v.\text{pid}, v.\text{pun}, v.\text{offset}, v.\text{len}) \) can be located via \( v_l.\text{tr} \). We use \( v_l \) and \( v_r \) to denote the left and right ends of node \( v \).

Since the left ends of all nodes are uniquely located and any character can be uniquely located relative to a node’s left end, characters in a model can also be regarded as uniquely located.

**Property 6.** Ordering of characters. (a) Intra-node: If \( v.\text{strInView} \) and \( \text{v.offset} \leq \text{pos} \prec \text{pos} < \text{v.offset} + \text{v.len} \), then \( \text{pos} \leq \text{pos} \).

Some particular nodes are used for the synchronization between the model and the view. Node \( \text{curr} \) and position \( \text{pos.curr} \) in \( \text{curr} \), define the current position of the model. It corresponds to the current position \( \text{curr.view} \) of the view. \( \text{render} \) and \( \text{render} \) mark the range of nodes that might need be rendered to synchronize the view with the model.

**Property 7.** Either (a) Empty range: \( \text{render} \) = \( \text{render} \) = nil and for all \( v \) in model, \( v.\text{rendered} \), or (b) Non-empty range: \( \text{render} \) = \( \text{render} \) = nil \land \text{render} \geq \text{render} \geq \text{render} \geq \text{render} \Leftrightarrow \text{render} \Leftrightarrow \text{render} \Leftrightarrow \text{render} \).

**Property 8.** Let \( v_i (i = 1, \ldots, n) \) be all the nodes left to \( \text{curr} \) and \( v_i.\text{strInView} \),

\[
\text{curr.view} = \begin{cases} 
\sum v_i \cdot \text{len} + (\text{pos.curr} - \text{curr.offset}) & \text{if } \text{curr.strInView} \\
\sum v_i \cdot \text{len} & \text{otherwise}
\end{cases}
\]

Finally, a peer maintains a log of operations that can be undone. A log entry has a reference to the leftmost node for ins) or element (for del or undo) of the operation in the model.
5. OPERATIONS AND UPDATES

A user may execute the following (user-oriented) operations in the view:

- \texttt{ins(pos, str)} inserts string \textit{str} at position \textit{pos}.
- \texttt{del(pos, len)} deletes \textit{len} characters right to position \textit{pos}.
- \texttt{undo(pid, pun)} undoes an operation identified by \textit{(pid, pun)} which can be \texttt{ins}, \texttt{del} or \texttt{undo}.

After an operation is executed in the view, it is enqueued in the peer’s \(Q_v\). Consecutive operations in the queue may be merged. For example, a sequence of character operations can be merged into a string operation. These operations are turned into the following (model-oriented) operations before they are dequeued for integration in the model.

- \texttt{move(\delta)} moves the current position a distance of \(\delta\) characters. If \(\delta\) is positive, move rightwards; otherwise, move leftwards.
- \texttt{ins(str)} inserts string \textit{str} at the current position.
- \texttt{del(len)} deletes \textit{len} characters right to the current position.
- \texttt{undo(pid, pun)} is the same as the user-oriented \texttt{undo} operation.

After a local operation is integrated in the model, a representation of the corresponding \textit{update} is enqueued in \(Q_{\text{out}}\) and the leftmost node or element of the operation is appended to the log.

An insertion update is represented with the \textit{id} of the deletion and the dependencies \texttt{dep}_I and \texttt{dep}_P of the newly inserted node, i.e. the place of the insertion.

6. MODEL-VIEW SYNCHRONIZATION

A user edits the document at the view of the local peer. The executed view operations are placed in \(Q_v\), after some conversion. Meanwhile, the peer may receive \(Q_{\text{in}}\) updates from remote peers. From time to time, the model and the view are synchronized.

Procedure \texttt{synch( fullsynch) }

\begin{verbatim}
while op ← \(Q_v\).dequeue() do
  \(Q_{\text{out}}\).enqueue(integrateLocal(op))
if fullsynch then
  while update ← \(Q_{\text{in}}\).dequeueReady() do
    integrateRemote(update)
render()
\end{verbatim}

Procedure \texttt{synch} synchronizes the view and the model. There are two options: either can a peer synchronize operations in both \(Q_v\) and \(Q_{\text{in}}\), or can it synchronize only operations in \(Q_v\).

Procedure \texttt{integrateLocal} integrates a local operation. The update representations of the integrated operations are enqueued in \(Q_{\text{out}}\), which are later broadcast to remote peers. Procedure \texttt{dequeueReady} dequeues from \(Q_{\text{in}}\) a remote update that is ready at this peer (Definition 5). Procedure \texttt{integrateRemote} integrates a remote update. Procedure \texttt{render} makes the effects of integrated remote operations available in the view. Because all integrated remote operations are rendered after a synchronization, there is no concurrent update in the model when local operations are integrated.

Local \texttt{undo} operations require immediate interaction with the model and need special handling. When a user issues an \texttt{undo} operation, the operation is enqueued to \(Q_v\) and Procedure \texttt{synch} is executed immediately (either full or local-only). Thus the \texttt{undo} operation is integrated into the model after all previously enqueued local operations. After Procedure \texttt{render} inside \texttt{synch} is done, the effect of the \texttt{undo} is shown in the view.

A peer may decide how often the model and the view are synchronized, with the restriction that every local \texttt{undo} operation must involve at least a local-only synchronization. Basically, with more frequent synchronizations, the view is more responsive to remote updates, at the cost of higher run-time overhead, and the user is more frequently distracted by the concurrent remote updates. A peer may also decide how often the updates of the integrated local operations are broadcast to remote peers.

6.1 Moving a distance and going to a node

To integrate executed local operations or render integrated updates to the view, positions in the model must correspond to the respective positions in the view. This is achieved through the \texttt{move} and \texttt{goto} procedures. Knowing a distance (i.e. number of characters) in the view, \texttt{move} places the current position of the model to the right position of the right node. Knowing a position of a destination node \(v\), \texttt{goto} finds the corresponding position in the view. Both procedures ensure that Property 8 holds.

Both the \texttt{move} and \texttt{goto} procedures traverse nodes via the \texttt{l-r} links and counts the lengths of the traversed nodes whose \textit{strInView} values are \texttt{true}. \texttt{goto} compares \textit{v.pos} with \textit{curr.pos} to decide which direction to go.

6.2 Model rendering

A rendering makes the effects of integrated updates appear in the view. According to Property 7, only nodes between \texttt{render} and \texttt{render}, (inclusive) need to be rendered. Suppose that \textit{render} \(\leq\) \textit{curr} \(\leq\) \textit{render}, and \textit{pos \texttt{curr}} is at the left end of \textit{curr}. Procedure \texttt{renderCurrRightward} renders \textit{curr} and places the current position to the left end of the node just right to \textit{curr}. To save space and keep readability, we do not enumerate all cases of rendering a node.

Procedure \texttt{renderCurrRightward()}

\begin{verbatim}
if ~curr.rendered then 
  if curr.visible \&\& ~curr.strInView then 
    view.ins(pos onView, curr.str); curr.strInView ← true 
  if ~curr.visible \&\& curr.strInView then 
    view.del(pos onView, curr.len); curr.strInView ← false 
  curr.rendered ← true
if curr.strInView then pos onView ← pos onView + curr.len 
  curr ← curr.r; pos curr ← curr.offset 
\end{verbatim}

curr.visible is calculated according to Definition 3. When \textit{curr} is involved in many concurrent \texttt{dels} and \texttt{undos}, calculating \textit{curr.visible} can be more costly than, say, checking \texttt{curr.strInView}. A node’s visibility is only calculated while the node is being rendered.
7. INTEGRATING LOCAL OPERATIONS

A model integrates a local ins operation by inserting a new node at an empty interval of the model.

**Definition 4.** An interval of a model is a pair \((v_l, v_r)\) of node ends, where \(v_l < v_r\). The interval is empty if \(v_l = v_r\).

Because node ends are always uniquely located in a model (Property 5), intervals are always uniquely located as well.

If \(pos_{curr}\) is in the middle of \(curr\), \(curr\) is split at \(pos_{curr}\) and the new node is inserted in between. Otherwise, there might be invisible nodes at the place of insertion. If this is the case, a particular new node is inserted in between. Otherwise, there might be invisible nodes at the right of the new node.

The interval at which the new node is inserted is the insertion dependencies \(dep_l\) and \(dep_r\) of the new node. If this node is subsequently split, all new nodes after splitting have the same insertion dependencies. When a local insertion is integrated, there is no concurrent remote operation integrated in the model (this is enforced by Procedure synch). Therefore, the interval at which the new node is inserted is empty, that is, it is defined by two adjacent nodes.

A model integrates a local del operation by associating del elements to the corresponding nodes whose strings are currently in the view. Nodes at deletion boundaries may be split.

A model integrates a local undo by associating undo elements to the corresponding elements (node for insertion, del for deletion and undo for undo). The nodes involved in the undo are marked as not rendered, and the render and render, are updated accordingly. Every local undo involves a synchronization between the model and the view, as described in Section 6.

8. INTEGRATING REMOTE UPDATES

A peer integrates a remote update only when the update is ready for integration at the peer. Otherwise, the update remains in \(Q_{ru}\).

**Definition 5.** A remote update is ready for integration at a peer when all its referenced node positions and elements are available in the model of the peer.

More specifically, a remote insertion is ready when the dependent node positions are available. A deletion is ready when all node positions encoded in the deletion update are available. An undo is ready when all nodes or elements of the operation it undoes are available. The involved nodes or elements may be split due to concurrent operations. The positions and elements are regarded as available both before and after the split. The ready-for-integration condition is less strict than “ready” or “happen-before” condition in the literature (such as [7, 19]), because only the node positions and elements that the update directly depends on are required to be available.

8.1 Insertion

Concurrent insertions may conflict with one another. It is crucial that the integration algorithm enforces the same \(\prec\)-order at all peers. The enforcement of the total order is based on the dependencies of the insertions.

**Definition 6.** Let \(v_a\) be a node of insertion \(ins_a\). \(ins_a\) inserts a node \(v_l\) of \(ins_a\) if (a) Direct dependence: there exists a node \(v_l\) of \(ins_a\) inside \((\bar{v}, \bar{v})\), such that \(v_l = v_a\) or \(v_l = v_a\) or \(v_l = v_a\); or (b) Indirect dependence: there is a third insertion \(ins_{\beta}\) such that \(ins_{\beta}\) insert-depends on \(ins_l\) at \((\bar{v}, \bar{v})\) and \(ins_l\) insert-depends on \(ins_{\beta}\).

Figure 3 shows an example of concurrent insertions. Figure 4 shows insert dependencies among these insertions at \((\bar{v}, \bar{v})\). The arrows depict direct insert dependencies. Note that \(\text{ins}(\text{"b"})\) insert-depends on \(\text{ins}(\text{"f"})\) at \((\bar{v}, \bar{v})\) but not at \((\bar{v}, \bar{v})\) or \((\bar{v}, \bar{v})\).

**Definition 7.** Two insertions \(ins_a\) and \(ins_b\) are dep-ordered, written as \(\text{ins}_a \prec \text{ins}_b\) if either there exists node \(v\) such that \(\text{ins}_a (\text{del}) = v\) and \(\text{ins}_b (\text{del}) = v\), or there exist nodes \(v_l\) and \(v_r\) such that \(\text{ins}_a (\text{del}) = v_l\), \(v_l \prec v_r\), and \(\text{ins}_b (\text{del}) = v_r\).

In Figure 4, \(\text{ins}(\text{"b"}) \prec \text{ins}(\text{"g"})\).

**Definition 8.** Two insertions \(ins_a\) and \(ins_b\) conflict at interval \((\bar{v}, \bar{v})\) if they are not dep-ordered, both are to be inserted at \((\bar{v}, \bar{v})\) and none of them insert-depends on the other at \((\bar{v}, \bar{v})\).

In Figure 4, \(\text{ins}(\text{"c"})\) conflicts with \(\text{ins}(\text{"d"})\) at \((\bar{v}, \bar{v})\). \(\text{ins}(\text{"e"})\) does not conflict with \(\text{ins}(\text{"d"})\) because \(\text{ins}(\text{"e"})\) insert-depends on \(\text{ins}(\text{"d"})\).

**Definition 9.** Two insertions \(ins_a\) and \(ins_b\) directly conflict at interval \((\bar{v}, \bar{v})\), if they conflict at \((\bar{v}, \bar{v})\) and there does not exist a third insertion \(ins_{\gamma}\) at \((\bar{v}, \bar{v})\) such that \(ins_a\) or \(ins_b\) insert-depends on \(ins_{\gamma}\).

In Figure 4, \(\text{ins}(\text{"x"})\) conflicts with \(\text{ins}(\text{"d"})\) at \((\bar{v}, \bar{v})\). However, \(\text{ins}(\text{"x"})\) does not directly conflict with \(\text{ins}(\text{"d"})\) at \((\bar{v}, \bar{v})\), because \(\text{ins}(\text{"d"})\) insert-depends on \(\text{ins}(\text{"i"})\).
Our algorithm enforces a \( \prec \)-order among directly conflicting insertions with a policy that is agreed upon at all peers. A commonly used policy is based on the \( \text{pid} \) of the original peers of the insertions. For instance, the insertion with smaller \( \text{pid} \) is placed to the left.

Integrating a remote insertion update is handled in iterations, as illustrated in Figure 5. To insert \( v_r \) at \((V_L, V_H)\) at Peer 2, we first find insertions at \((V_L, V_H)\) that directly conflict with \( v_r \). In our example, this is the insertion of \( v_f \). Since \( \text{ins}(“F”) \) has \( \text{pid} \) 4 and \( \text{ins}(“x”) \) has \( \text{pid} \) 3, \( v_r \) is to be inserted to the left of \( v_f \) and \( v_r \) is then to be inserted at \((V_L, V_H)\) in the next iteration. The directly conflicting insertions are then \( \text{ins}(“bb”) \), \( \text{ins}(“c”) \) and \( \text{ins}(“d”) \). \( v_r \) is then inserted at \((V_L, V_H)\) according to the \( \text{pid} \) of the originating insertions. At \((V_L, V_H)\) there is no insertion conflicting with \( \text{ins}(“x”) \). Therefore \( v_r \) is finally inserted between \( v_f \) and \( v_d \).

This process is in essence the same as in [11]. Notice that deletion and node splitting have no effect with respect to the ordering of the insertions.

**Procedure insertRemote** \( (v_{\text{ins}}, v_L, v_R) \)

\[
v \leftarrow v_{\text{ins}}.r
\]

while \( v \neq v_{\text{ins}} \) do

if \( v \text{.dep} \cdot \text{pos}^{\text{eff}} \leq v_{\text{ins}} \cdot \text{pos}^{\text{eff}} \land v \text{.dep} \cdot \text{pos}^{\text{eff}} \geq v_{\text{ins}} \cdot \text{pos}^{\text{eff}} \) then

--- \( v \) directly conflicts with \( v_{\text{ins}} \)

if \( v_{\text{ins}} \cdot \text{pos} < v \cdot \text{pos} \) then \( v_{\text{ins}} \leftarrow v \) --- \( v_{\text{ins}} \) < \( v \)

else \( v_{\text{ins}} \leftarrow v \) --- \( v_{\text{ins}} \) > \( v \)

\[ v \leftarrow v_{\text{ins}}.r \] --- new iteration

else \( v \leftarrow v_{\text{ins}} \) --- same iteration

insertBetween \( (v_{\text{ins}}, v, v_{\text{ins}}) \)

Procedure \( \text{insertRemote} \) is an improvement to the process just described. It inserts \( v_{\text{ins}} \) between \( v_L \) and \( v_R \). To integrate a remote insert update \( v_{\text{ins}} \), we start with \( v_{\text{ins}} \cdot \text{dep} \) as \( v_L \) and \( v_{\text{ins}} \cdot \text{dep} \) as \( v_R \). Thus \( v_{\text{ins}} \) does not insert-depend on any insertion at \((V_L, V_H)\). The loop walks the node \( v \) through the nodes between \( v_L \) and \( v_R \). If \( v \) does not insert-depend on any insertion at \((V_L, V_H)\), \( v_{\text{ins}} \) and \( v \) directly conflict at \((V_L, V_H)\). We enforce the order between \( v_{\text{ins}} \) and \( v \) according to their \( \text{pid} \) values and adjust \( v_L \) or \( v_R \) accordingly for the next iteration. We use the \( \text{pos}^{\text{eff}} \) values to figure out whether \( v \) insert-depends on any node at \((V_L, V_H)\). If \( v \cdot \text{dep} \cdot \text{pos}^{\text{eff}} > v_{\text{ins}} \cdot \text{pos}^{\text{eff}} \) or \( v \cdot \text{dep} \cdot \text{pos}^{\text{eff}} < v_{\text{ins}} \cdot \text{pos}^{\text{eff}} \), then \( v \) insert-depends on a node between \( v_L \) and \( v_R \). (Without the use of \( \text{pos}^{\text{eff}} \), another loop is needed for this purpose [11].) If \( v \) insert-depends on some node between \( v_L \) and \( v_R \), \( v \) moves to the next node and the loop goes on for the same iteration.

8.2 Deletion and undo

Integrating a remote \( \text{del} \) or \( \text{undo} \) simply associates the corresponding \( \text{del} \) or \( \text{undo} \) elements in the model data structure.

A node of a deletion is represented as \((\text{pid}, \text{pun}, \text{offset}_{\text{old}}, \text{offset}_{\text{new}}, \text{len})\) in an undo message. If node \((\text{pid}, \text{pun}, \text{offset}_{\text{old}})\) has already been split by a concurrent operation at a different position, the integrating peer can use \((\text{pid}, \text{pun}, \text{offset}_{\text{old}})\), \text{offset}_{\text{new}}\) and \text{len} to find the correct node.

When the operation that a remote \( \text{undo} \) undoes has already been undone by a concurrent \( \text{undo} \), the corresponding element is already associated with an \( \text{undo} \) element. In this case, the id of the remote \( \text{undo} \) is inserted to the \text{ties} element of that \( \text{undo} \) element.

Notice that concurrent deletions of the same sub-string have an accumulative effect (i.e. the overlapping sub-string is deleted multiple times), while concurrent \( \text{undos} \) have an idempotent effect. For either of them, integrating concurrent operations in different order results in the same visibility of the corresponding nodes.

9. CORRECTNESS

We consider two correctness criteria, intention preservation and convergence, as defined in [19].

9.1 Intention preservation

Intention preservation [19] requires that, for any operation \( \text{op} \), (1) the effects of executing \( \text{op} \) at all peers are the same as the intention of \( \text{op} \), and (2) the effect of executing \( \text{op} \) does not change the effects (i.e., intentions) of independent operations.

Intention is not formally defined in [19] and is open for different interpretations. This is one of the reasons why correctness of OT algorithms are difficult to be proven formally [7]. A commonly adopted interpretation of intention of an insertion is the effect-relation order introduced by the originating intention [7, 11].

In our work, the intention of an operation is decided at the view of the originating peer. More specifically, an intention is between two characters at the current position at the time of insertion; a deletion removes a sub-string of characters from the view; undo of an intention removes the inserted characters from the view; undo of a deletion makes the removed characters re-appear in the view and the positions of the re-appeared characters must preserve the intentions of the corresponding insertions.

First, we verify that integrating an operation preserves its intention.

For insertion, we consider the ordering of nodes. For deletion and undo, we only consider the visibility of nodes, because they do not change the ordering of nodes.

Because splitting a node does not change the ordering among existing nodes or the visibility of nodes, it does not change the intention of any operation.

Integrating a local insertion inserts the new node between the two nodes of the visible characters at the current position. Placing the invisible nodes to the left of the newly inserted node does not change the intention. Integrating a remote insertion also inserts the new node between the two nodes of the same characters. The existence of concurrent insertions does not change the intention. Integrating a local and remote deletion makes the same set of characters invisible in the view, thus preserving its intention.

Undo of a local or a remote insertion makes the characters of the insertion invisible. Undo of a local or remote deletion, or redo of an insertion, makes the characters that are not deleted by other deletions visible again. The intention of the original insertion is preserved after the undo or redo, because the characters are still in the same order.

Next, we verify that integration of an operation does not change the intentions of other operations.

No procedure switches the order of nodes. Therefore the intention of an insertion will never be changed by the integration of any other operation.

Undoing a deletion brings the deleted characters back in the view only when the insertions of the corresponding characters are not undone and the characters are not deleted by any concurrent overlapping deletion. Therefore undo of a deletion does not change the intention of the undo of any insertion and any other deletions. This is in contrast with all related work that defines the effect of concurrent deletions of the same character as a single deletion. Undo of one deletion thus changes the intentions of all these concurrent deletions. Two concurrent deletions, despite overlaps, are different operations. Undoing one of them should not change the intention of the other.

On the other hand, concurrent \( \text{undos} \) of the same operation are considered as a single undo, because they are always defined on the same operation.
9.2 Convergence

Convergence [19] requires that, when the same set of operations have been executed at all peers, all copies of the shared document are the same.

For insertions, it is crucial to show that all peers enforce the same \(\prec\)-order among the characters. (Although the key idea of the algorithm was published in [11], there has not been a formal proof of its correctness.)

Due to Procedure \textit{synch} in Section 6, there is no conflicting insertions during integration of local insertions. Therefore, we only need to verify that integrating remote insertions ensures a globally unique \(\prec\)-order among concurrent insertions.

To see that all peers enforce the same \(\prec\)-order among insertions, we verify that any peer enforces the same \(\prec\)-order as an ideal peer, \textit{peer0}, where all insertions are available. \textit{peer0} integrates insertions in iterations. An iteration consists of insertions that insert-depend on insertions of earlier iterations. These insertions are separated into groups by the nodes in earlier iterations. In every group, insertions directly conflict with each other between two nodes of earlier iterations. If for node \(v\) of an insertion, \(v.\text{itr}\) is the iteration in which the insertion is integrated at \textit{peer0}, we have \(v.\text{itr} = \max(v.\text{dep}_1, v.\text{dep}_2, v.\text{dep}_3, v.\text{itr}) + 1\). Here and in the rest of this section, \(v.\text{dep}_1\) or \(v.\text{dep}_2\) is used to denote the node containing that node end, when this does not cause confusion from the context.

In Figure 3, assume the insertions in Iteration 0 are \(\{"a", "h"\}\), then the insertions in Iterations 1, 2, 3 are \(\{"f", "x"\}\), \(\{"bb", "c", "d", "g", "y"\}\) and \(\{"e"\}\).

\textbf{Lemma 1.} The \(\prec\)-order of all insertions at the ideal peer is unique.

\textbf{Proof.} We prove with induction.

At iteration 0, all conflicting insertions are directly conflicting with each other. Their \(\prec\)-order is decided by the \textit{puns} of the originating peers and therefore is unique.

Assume at iteration \(k\) the \(\prec\)-order of all insertions is unique. At iteration \(k+1\), we show, in the following two steps, that the \(\prec\)-order of all insertions is also unique:

(a) Any insertion of iteration \(k+1\) is uniquely placed among the nodes of iteration \(k\) or below.
(b) For all insertions of iteration \(k+1\) that are to be placed at the same place among the nodes of iteration \(k\) or below, they are uniquely ordered.

To show (a), we will show that if a node \(v\) of iteration \(k+1\) were inserted without the existence of any other node of the same iteration, it would have been placed immediately between nodes \(v_1\) and \(v_2\). Then the insertion of node \(v\) (also of iteration \(k+1\)) prior to \(v\) would cause \(v\) to be placed outside of \(v_1\) and \(v_2\). We show this with contradiction.

First, we know that either (i) \(v.\text{dep}_1 = v_1\), or (ii) \(v\) directly conflicts with \(v_1\) during the last iteration of Procedure \textit{insertRemote}, because otherwise \(v \prec v_1.\text{dep}_1\) or \(v_1.\text{dep}_2 \prec v\) and \(v_1\) would not be the immediate left neighbor of \(v\) as the final result of the procedure. Therefore, either (i) \(v_1 \prec v\) or (ii) \(v.\text{dep}_1 \prec v \prec v_1.\text{dep}_1\) and \(v.\text{pun} > v_1.\text{pun}\).

The only possibility that the insertion of \(v\) prior to \(v\) makes \(v\) be placed left to \(v_1\) is \(v \prec v_1\). This is impossible with case (i) above. Now consider case (ii). Because \(v\) directly conflicts with \(v_1\), \(v.\text{pun} < v_1.\text{pun}\). However, \(v\) is placed left to \(v_1\) either when \(v' \prec v_1.\text{dep}_1\) or when \(v\) directly conflicts with \(v_1\) and \(v.\text{pun} > v_1.\text{pun}\).

In the former case, \(v \prec v_1\) \(v_1.\text{dep}_1\) contradicts with \(v_2.\text{dep}_1\) \(v\). In the latter case, \(v.\text{pun} < v_1.\text{pun} < v_1.\text{pun}\) contradicts with \(v.\text{pun} > v_1.\text{pun}\).

In Figure 6, assume the insertions in Iteration 0 are \(\{"d", "g", "y"\}\) and \(\{"e"\}\). Then the insertions in Iterations 1, 2, 3 are \(\{"f", "x"\}\), \(\{"bb", "c", "d", "g", "y"\}\) and \(\{"e"\}\).

\textbf{Theorem 1.} When all peers have integrated the same set of insertions, the \(\prec\)-order of the integrated insertions are the same at all peers.

\textbf{Proof.} Assume that at a peer, a pair of insertions are integrated out of the ideal iteration order. We will show that this peer enforces the same \(\prec\)-order as \textit{peer0}. Extending this to any pair of out-of-order insertions, we can conclude that a peer enforces the same \(\prec\)-order as \textit{peer0}.

Given nodes \(v_1\) and \(v_2\) and \(v_2.\text{itr} > v_1.\text{itr}\) (Figure 6). Assume \(v_1\) and \(v_2\) may be integrated out of the ideal order. Because \(v_1\) may be integrated after \(v_2\), \(v_2\) does not insert-depend on \(v_1\) (Definition 5). We show that integrating \(v_2\) prior to \(v_1\) enforces the same \(\prec\)-order as integrating \(v_1\) after \(v_2\).

There are only three possible cases:

1. The intervals \((v_1.\text{dep}_1, v_1.\text{dep}_2)\) and \((v_2.\text{dep}_1, v_2.\text{dep}_2)\) do not overlap, (including \(v_2.\text{dep}_1 = v_1.\text{dep}_1\) and \(v_2.\text{dep}_2 = v_1.\text{dep}_2\)).
2. \(v_1.\text{dep}_1\) is inside \((v_2.\text{dep}_1, v_2.\text{dep}_2)\).
3. \(v_1.\text{dep}_1\) is outside \((v_2.\text{dep}_1, v_2.\text{dep}_2)\) and \(v_2.\text{dep}_1\).

In case 1, either \(v_1 \prec v_2\) or \(v_2 \prec v_1\) (Definition 7), thus integrating \(v_1\) (and \(v_2\) respectively) does not involve \(v_1\) (and \(v_2\) respectively). Therefore integrating \(v_1\) after \(v_2\) results in the same \(\prec\)-order as integrating \(v_1\) before \(v_2\). In case 3, Procedure \textit{insertRemote} in Section 8.1 first resolves \(\prec\)-order of \(v_1\) against \(v_2.\text{dep}_1\) and/or \(v_2.\text{dep}_2\). If \(v_1\) is placed outside \((v_2.\text{dep}_1, v_2.\text{dep}_2)\), like case 1, the rest of the procedure for integrating \(v_1\) will never involve \(v_2\).

Now consider that \(v_1\) inside \((v_2.\text{dep}_1, v_2.\text{dep}_2)\) (case 2 and the rest of case 3). If \(v_1\) directly conflicts with \(v_2\) at \((v_2.\text{dep}_1, v_2.\text{dep}_2)\), their ordering is decided by their \textit{pid} values, which is independent of the temporal order of their integration.

If \(v_1\) does not directly conflict with \(v_2\) at \((v_2.\text{dep}_1, v_2.\text{dep}_2)\), according to Definition 9, either \(v_1.\text{dep}_1\) or \(v_1.\text{dep}_2\), or both, is inside \((v_2.\text{dep}_1, v_2.\text{dep}_2)\). Without loss of generality, assume that \(v_1.\text{dep}_1\) is inside \((v_2.\text{dep}_1, v_2.\text{dep}_2)\). The ordering between \(v_1.\text{dep}_1\) and \(v_2.\text{dep}_1\) is resolved first (when \(v_2\) was inserted). If \(v_2 \prec v_1.\text{dep}_1\), we have \(v_2 \prec v_1\) and integrating \(v_1\) does not involve \(v_2\). If \(v_1.\text{dep}_1 \prec v_2\), then we consider the cases where \(v_1\) directly conflicts with \(v_2\) or not at \((v_1.\text{dep}_1, v_2.\text{dep}_2)\) and can similarly verify that integrating \(v_2\) prior to \(v_1\) enforces the same ordering as integrating \(v_2\) after \(v_1\).

Therefore in all three possible cases, integrating \(v_2\) prior to \(v_1\) enforces the same \(\prec\)-order as integrating \(v_2\) after \(v_1\).

The convergence of deletions and \textit{undos} is straightforward. Because the visibility of nodes is independent on the order of operation integration, when the same set of operations are integrated at
Real-time collaborative editing requires high responsiveness to local user operations, since system responsiveness has clear effects on human performance. For example, [9] showed that at response time of 75ms, user performance becomes unstable, and at 225ms, user performance is degraded substantially. The effects of responsiveness to remote operations, on the other hand, depends on the application and the context of the application. For instance, distributed pair programming is more demanding on responsiveness to remote user operations than collaborative writing of a technical report, particularly when the co-authors focus on different sections of the report.

The response time of local view operations is an important measure of an editor’s local responsiveness. Except selective undo, all view operations are executed completely in the view. Their performance therefore are nearly the same as a single-user editor.

Local view operations are executed only when system resources (CPU, memory etc.) are available. When the system is heavily loaded, the user will experience low responsiveness even though the local view operations take very short time. Therefore responsiveness is dependent on the overall performance of the editor, including the more expensive model operations.

In this section, we first discuss time complexity of the algorithms and space overhead of the data structure, and then present experimental results.

### 10.1 Time complexity

Table 1 shows the time complexity of the different procedures. $m$ is the distance of a move, i.e., the number of nodes a move traverses. $r$ is the size of the region to be rendered, i.e., the number of nodes in the region. $l$ is the size of an operation, i.e., the number of nodes that the operation involves. $k$ is the number of conflicting insertions. $s$ is the span of an operation, i.e., the number of nodes between the leftmost and rightmost nodes of the operation, including the nodes not belonging to the operation. For example, for a local insertion, $s$ is the number of invisible nodes between two visible nodes at the place of an insertion;

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>move</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>local ins</td>
<td>$O(s)$</td>
</tr>
<tr>
<td>local del</td>
<td>$O(s)$</td>
</tr>
<tr>
<td>local undo</td>
<td>$O(m + s)$</td>
</tr>
<tr>
<td>render</td>
<td>$O(m + r)$</td>
</tr>
<tr>
<td>remote ins</td>
<td>$O(k^2)$</td>
</tr>
<tr>
<td>remote del</td>
<td>$O(l)$</td>
</tr>
<tr>
<td>local undo</td>
<td>$O(l)$</td>
</tr>
</tbody>
</table>

It can be seen that a local undo can be an expensive operation, because the current position must first be moved to the node of the operation and the effect of the undo must then be rendered. For a local-only synch, the span of the undo is the size of the range for rendering. $m$ and $s$ can be considerably larger than $l$, the size of the operation to be undone.

To render the integrated remote updates, $m$ nodes are first traversed to go to the edge of the region for rendering. This is costly when $m$ is large. On the other hand, when $m$ is large, the concurrent remote updates are far from the document area where the local editing is focused on. In such situation, less frequent full synchronization would reduce the overall overhead of the system.

The number of concurrent insertions $k$ is typically small. Although in the worst case it may take up to $k^2$ steps to integrate a remote update, the range $(v_r, v_u)$ in Procedure insertRemote typically shrinks very quickly.

In related work that do not support string-wise operations, $l = 1$, but the number of operations are much larger. This implies a larger number of invocations to the corresponding procedures, longer operation histories and larger number of network messages.

### 10.2 Space overhead

Materializing relations among operations incurs space overhead. The materialized relations contribute not only to the correctness of the algorithms (for example, an undo always refers to the same characters of the operation it undoes), but also to the overall performance (OT approaches, for example, derive the relations through operation transformation every time the relation is needed). This space overhead, however, may not be larger than that in related work. Notice that the space overhead is shared not only by all the characters of a node, but also by all operations associated with the node. In related work, every single operation of every single character has its own space overhead for meta-data, either in terms of state vector or operation context in OT approaches, or in terms of character identifiers in CRDT approaches. To accommodate this overhead, [22] for instance, supports line-wise, rather than character-wise, operations.

To make the analysis more specific, let us assume that for a node, the size of the fixed part (pid, pun, offset, rendered, dep, depr, pos, l, r, il, ir) is $F$, it represents a string of $s$ characters, has $d$ del elements of size $D$ and a chain of $u$ undo elements of size $U$. The total size of the node is $N = F + d \cdot D + u \cdot U$.

Here we ignore the space taken for the character string, because the same space is taken in all approaches.

Now, consider the corresponding information stored in an operations history in an OT system that only supports character operations. Assume that the size of an operation record in the history is $R$ (a record contains information like state vector or context as well as some other meta data). The information contained in the above-mentioned node includes $s$ single-character insertions, of each character, $d$ deletions and $u$ undos. The total size of all these operation records is $R_N = s \cdot R + s \cdot d \cdot R + s \cdot u \cdot U$.

To make the comparison more straightforward, assume that $F \approx D = U$. The difference between the two is $N/R_N = F/(s \cdot R)$. $F$ and $R$ are typically comparable. When $s$ is large, the space overhead of our approach is a small fraction of a character-wise OT approach.

### 10.3 Experiments

We have implemented the core algorithms in Emacs Lisp, aiming at supporting group editing in a widely used open-source editor. Our experiment for performance study is based a trace of operations for editing a program in a week’s period. We then re-play the operations in different settings to measure the time of different procedures presented in Sections 6, 7 and 8. The measurement was taken under GNU Emacs 24.3.1 running in 32-bit Linux 3.10.5-ARCH on a 7-year old ThinkPad T61p (2007 model) with 2.2GHz Intel Core2 Duo CPU T7500 and 2GB RAM.

The trace first captures the editing operations in view, as presented in Section 5. The captured view operations are then aggregated and converted into model operations. There are different ways of aggregating the view operations. In the experiments, we aggregated the operations as far as possible. That is, consecutive insertions and deletions are aggregated into a single operation, until the cursor moves away from the boundary of the current inserted or deleted string. Figure 7 shows the number of model operations and their lengths (numbers of characters) obtained from the trace.
In the first experiment, there are two peers. The first peer, the local peer, re-plays the traced operations as local operations. To measure the time for selective undo, after re-played all traced operations, the peer undoes all these operations in reverse order. The updates for the integrated operations are then sent to the second peer, the remote peer, which integrates the received remote updates.

Let us first look at Figure 8. The y-axis is the time for running Procedure $\text{sync}$ at the remote peer. The x-axis is the time at which the procedure is called. Because the experiment only re-plays the traced operations, the x-axis values are of no interest and thus are not shown. For the vast majority of times, Procedure $\text{sync}$ takes less than 1 ms. There are however occasions where it takes over 20 ms. It turns out, with closer investigation, that it is the garbage collection of Emacs Lisp that contributes to these long delays. In what follows, we will show the time of different procedures without those extra delays caused by garbage collection of Emacs Lisp.

Figure 9 shows the execution time of integrating the different operations. An overall observation is that the integration time of all operations is almost independent of the length of the operation history.

In general, integrating local operations takes longer time than integrating remote updates. One reason is that the time for integrating a local operation includes the time for encoding the update into its JSON representation, whereas the time for integration a remote update does not include the time for decoding the JSON representation, because the decoding has already been done for the testing of the ready-for-integration condition.

The time for integrating remote insertions seems to be more fluctuant than for integrating local insertions. In fact, if we “zoom in” with the same resolution, the time for integrating local insertions is at least as fluctuant. The fluctuation is largely due to the maintenance of Property 3, that is, to re-assign the $\text{pos}^{\text{df}}$ values among a set of nodes, when a node is inserted between two nodes, but there is no new $\text{pos}^{\text{df}}$ value available at the place of insertion. In addition, the numbers of invisible nodes at the places of insertion (i.e. the span $s$) also cause the fluctuation of the time for integrating local insertions.

Integrating a local deletion can take much longer time (up to 15 ms) than integrating a remote deletion, because the span $s$ of a deletion could be much larger than its size $l$ (there may be nodes that are already invisible inside the region of the deletion). Notice that the shapes of the plots indicate a strong correlation between the spans and sizes of deletions.

Integrating a local selective undo could be expensive, because it involves at least a partial model-view synchronization. As a significant part of the synchronization, the current position must go to the place of the undo. On the other hand, integrating a remote selective undo is very fast. The plot for remote undo can even show the tiny difference between the time for undoing insertions and deletions.

Figure 10 shows the time related to model-view synchronization. Procedure $\text{move}$ is only called at the local peer. Procedures $\text{goto}$ and $\text{render}$ are called at both peers. At the local peer, they are only called during the integration of $\text{undo}$ operations and their time is already shown in Figure 9. Therefore Figure 10 only includes the time for Procedures $\text{goto}$, $\text{render}$ and $\text{sync}$ at the remote peer. Furthermore, these three plots can be regarded as been composed of two halves: the left half for integrating remote insertions and deletions and the right half for integrating remote $\text{undos}$.
could not observe clear impact of the number of operations (i.e. the
insertion (every insertion is done three times and undone twice), but we
ment the number of updates are many times as in the first experi-
0.05 ms to integrate a remote insertion. Notice that in this experi-
for rendering the updates. Most of the time it still takes less than
0.1 ms to render the updates are due to deletions with large spans.
Overall, we find the experimental results consistent with the the-
oretical time complexity presented in Subsection 10.1. The exper-
iments indicate that the distances of moving the current positions
m and the spans of operations s (in particular, some deletions may
have very large spans) are the dominant contributors to the observ-
able (in plots, not by end-users) long delays. Local selective undos
are generally expensive because each of them has to involve a (par-
tial) model-view synchronization. On the positive side, the sizes
of operations l are only secondary to the overall performance. The
performance is almost independent of the length of operation his-
tory (except that garbage collection of Emacs Lisp may take longer
time when there are more data elements in the runtime stack). With
very few exceptions (mostly deletions with very large spans), all
procedures finish under one millisecond. On the other hand, the
garbage collection of Emacs Lisp may take over 20 ms and still
hardly any end-user ever notices the delay. This strengthens our
confidence that even on an old laptop, our algorithms can provide
sufficient responsiveness to end-users.

11. OPEN ISSUES
As the performance of the approach depends on the spans of op-
erations and the number of nodes a move or goto traverses, it is
 desirable to suppress the model to reduce the number of nodes in
the data structure. This may include discarding invisible nodes and
combining adjacent visible nodes.
An invisible node plays two roles. (1) The node may become
visible again when an operation associated with the node is undone.
(2) The node is a landmark, i.e. another node insert-depends on it.
If we assume that (1) max L last operations originated at a peer
can be undone, and (2) all updates older than L are guaranteed to
have been delivered at all peers, then we can, by carefully maintain-
ing the right visibility of nodes, discard all del and undo elements
older than p.pun - L, where p.pun is the latest known pun of peer
p. We may also discard str of invisible nodes older than p.pun - L.
We may then combine nodes of the same insertions after the discard
of old elements. All this can be done locally at individual peers.
However, a peer cannot completely get rid of invisible nodes as
they can be landmarks. How to effectively suppress the data struc-
ture is still a challenging issue.
Another issue is session management. When a user joins a editing session or resumes a suspended one, the latest state of the model must be established at the peer. Materialization of operation relations makes session management more complicated.

Yet another important issue, not specific to our approach, is how to practically resolve conflicts. Currently, all approaches are based on the theoretical correctness, i.e. intention preserving and convergence. However, that a solution is theoretically correct does not imply that it is also practically or semantically correct. For example, if two peers both change the word “he” to “she”, the end result is “sshe”, which is theoretically correct but practically wrong.

12. CONCLUSION

The work presented in this paper advances the state-of-the-art of real-time decentralized group editing by supporting both string-wise operations and selective undo. The approach combines and extends the strengths of operation transformation (OT) and commutative replicated data type (CRDT) approaches by materializing operation relations in a data structure. This contributes not only to more straightforward enforcement of correct execution, but also to better runtime performance. This, however, makes some tasks more complicated, such as session management and garbage collection. We have proved the correctness of the approach using the classical correctness criteria intention preservation and convergence. We have also analyzed the complexity of the algorithms and verified the analysis with experiments. The time complexity of the algorithms is independent of the lengths of operation histories. That is, the execution time for executing and integrating various operations does not constantly grow over time. The experimental result is consistent with the analysis and indicates that the approach provides sufficient responsiveness to end-users.

13. REFERENCES