2009 15th International Conference on Parallel and Distributed Systems

An Optimized String Transformation Algorithm for Real-Time Group Editors

Bin Shao
School of Computer Science
Fudan University, China
binshao@fudan.edu.cn

Du Li
Nokia Research Center
Palo Alto, CA, USA
lidu008@gmail.com

Ning Gu
School of Computer Science
Fudan University, China
ninggu@fudan.edu.cn

***Abstract*—Operational transformation (OT) is an optimistic consistency control method for supporting collaboration over high-latency networks. The technique lies in the heart of many recent products such as Google Wave. It replicates the shared data and allows the users to concurrently modify any part of a shared document in a nonblocking manner. Most of the published results only support two characterwise primitives and take $O(|H|^2)$ or even longer time to integrate one remote characterwise operation, where H is the operation history. However, as the history grows long and operations are integrated in batches, the high complexity can make an algorithm easily exceed the 100 ms responsiveness threshold that is critical for interactive applications. This paper proposes a new OT algorithm that supports string primitives and reduces the time complexity to $O(|H|)$. The result can be used in a range of parallel and distributed applications that can be abstracted as realtime group editors.**

***Keywords*-Collaborative Systems; Data Consistency; Group Editing; Operational Transformation**

I. Introduction

A fundamental challenge in supporting collaboration over wide-area networks is high communication latencies. Data replication is a common approach to achieving high availability and performance in the wide area [9]. By replicating shared data at multiple sites, users can quickly access the shared data even when remote replicas are unavailable, e.g., due to site failure and network partition.

Operational transformation (OT) is a lock-free, nonblocking data replication technique that was originally motivated in the context of group editing [2]. The technique has been implemented in many products including CoWord [15], ACE, Gobby, SubEthaEdit, and most recently Google Wave. In an OT-based group editor, operations are first applied on the local replica as soon as generated and then propagated to remote sites in the background; remote operations are transformed before execution such that consistency can be maintained without a total order of operations. As a result, the local response time is not sensitive to networking latencies and the group editor can be as responsive as a single-user editor; users are allowed to modify any part of the shared data in parallel without blocking each other.

A large number of OT algorithms [4], [8], [12], [13], [17] have been proposed. To the best of our knowledge, the vast majority of published OT algorithms take $O(|H|^2)$ or even longer time to integrate one remote operation [5], where H is the operation history. Except for [14], they generally only support two characterwise primitives, insert and delete, which implies a practicality gap when supporting the commonplace string operations. In addition, for better utilization of system resources, operations are usually propagated and integrated in batches [7]. The consequence is that the history can grow so fast that even integrating one operation may exceed the 100 ms interactivity requirement [5]. As a result, previous works have to keep the history short enough to warrant interactivity. This is achieved by periodically and frequently garbage-collect operations in the history, which could be an expensive process itself [6], [14].

This paper presents a novel OT algorithm that extends the latest theoretical result in this area, Admissibility-Based Transformation or ABT [8], [6], to support efficient stringwise operations. Due to its focus on correctness, ABT only considers two characterwise primitive operations with time complexity $O(|H|^2)$. The new algorithm is called ABT-String Optimized or ABTSO. By supporting stringwise operations and improving the time complexity to $O(|H|)$, ABTSO can ensure the 100ms interactivity requirement even with a much longer history. This is achieved by keeping history operations in a special order called the operation effects relation [4], [8], [6], which is conceptually the relative position of objects inserted or deleted by the operations. Our experimental results will show that the optimized stringwise algorithm is much more efficient than its characterwise and unoptimized stringwise counterparts.

The rest of this paper is organized as follows. The next section introduces the background. Section III presents the ABTSO algorithm. Section IV analyzes complexities and presents experimental results. Section V compares related works. Finally, Section VI concludes.

II. Background and Notations

We first use a simple scenario that actually happened in one of the author's software project to show how an OT-based system works. Suppose that two users, Alice and Bob, collaboratively maintain a shared document containing a list of modules in a system they are developing together.

1521-9097/09 $26.00 © 2009 IEEE
DOI 10.1109/ICPADS.2009.72

IEEE computer society

Authorized licensed use limited to: Beijing Jiaotong University. Downloaded on November 11,2022 at 03:32:18 UTC from IEEE Xplore. Restrictions apply.

Initially, there is only one module named "email" in the list. The document is replicated so that the users work in parallel on different modules. Let the first position of a string be zero. Alice extends the list to "chat,email" by operation o_A = ins(0, "chat,") to add her module "chat". Concurrently, Bob extends the list to "email,calendar" by operation o_B = ins(5, ",calendar"). Now the two sites diverge.

When Alice receives o_B, if it were executed as-is, then Alice's document would become "chat,,calendaremail". The basic idea of OT is to transform o_B in this situation such that its resulting form can be safely executed in current state of the shared document "chat,email". Considering that o_A has inserted a string "chat," to the left of the intended position of o_B, we should shift the position of o_B by the length of that string, resulting in o'_B = ins(10, ",calendar"). Execution of o'_B on Alice's document yields "chat,email,calendar". Meanwhile, Bob receives o_A and executes it as-is, also yielding "chat,email,calendar". In this case, the position of o_A is not shifted because the execution of o_B on its right side does not invalidate its position. Now the two sites converge with the desired result.

From the above scenario, we know that ensuring a total order of operations (e.g., o_A followed by o_B) can achieve convergence but the result may not be what the users want. The power of OT lies in that it not only achieves convergence but also preserves operation intentions [14] or the operation effects relation [4], [8] without relying on a total order of execution. Any pair of operations can be transformed to commute in a well-design OT algorithm.

Suppose that there are a number of participants in a collaborative session. Each participant is called a site which is identified by a unique site id. All sites begin with the same copy of the shared data which is abstracted as a string of characters. Objects are identified by their positions in the string with the first position being zero. An operation o is represented as a quintuple $(id, type, pos, str, vt)$, where id is identifier of the site that generates o; $type$ is the operation type, either ins or del; pos is the position in the string where o is executed; str is the string to be inserted or deleted; and vt is the vector timestamp when o is generated. An operation can be abbreviated as $ins(pos, str)$ and $del(pos, str)$.

As shown in [14], when a stringwise deletion is transformed with another insertion or deletion, the deletion will be split into two or more sub-operations. The handling of deletion splits is very tricky and the discussions would be complicated. This paper simplifies presentation by avoiding deletion splits. Using notation $|s|$ for the length of string s, we keep $|o.str| = 1$ when o is a deletion.

In distributed systems, vector timestamps are widely used for determining the happens-before ($\rightarrow$) and concurrent ($\|$) relations between operations [2], [3]. Given any two operations o_1 and o_2, their relation is denoted as $o_1 \rightarrow o_2$ if o_1 happens before o_2; their relation is denoted as $o_1 \parallel o_2$ if neither $o_1 \rightarrow o_2$ nor $o_2 \rightarrow o_1$. An operation o is *causally-ready*, if all operations that happen before o have been executed at the site in question.

Following [13], an operation o's *definition state* $dst(o)$, is the state in which $o.pos$ is defined. Given any two operations o_1 and o_2: if dst(o_1) = dst(o_2), then they are *contextually equivalent*, denoted as $o_1 \sqcup o_2$; if o_2's position is defined in the immediate state of applying o_1, then o_1 and o_2 are *contextually serialized*, denoted as $o_1 \mapsto o_2$.

Any two executed operations can be ordered by their *effects relation* $\prec$, which conceptually is the relative position of the objects the two operations insert or delete. In the above example, $o_A \prec o_B$ because the relative position of o_A's effect "chat," precedes that of o_B's effect ",calendar". The concept is rigorously defined in [8], [6].

An operation sequence is a list of operations that are contextually serialized. An empty sequence is denoted as $[]$. An operation sequence sq that has n elements is denoted as $sq = [o_1, o_2, \ldots, o_n]$. Operator $\cdot$ concatenates two sequences, or a sequence and an operation.

III. The ABTSO Algorithm

This section will first show the structure of the ABTSO algorithm and then discuss the functions involved.

Each site maintains an operation log H which logs operations that have been executed at the current site. Similarly to ABT [8], ABTSO maintains H as a concatenation of two operation sequences, H_i and H_d, where H_i is the *insertion* log and H_d is the *deletion* log. The difference is mainly that in ABTSO operations in each subsequence are in their effects relation order, while in ABT they are in execution order. As a result, the time complexity of ABT is $O(|H_i|^2 + |H_d|)$ whereas that of ABTSO is $O(|H|)$.

In addition to the operation log H, each site also maintains a receiving queue RQ which keeps the operations received from other sites. A state vector sv indicates the number of operations from every site that have been executed at current site. sv is a list of N integers, where N is the total number of sites in the system.

The overall control procedure of any site j is as shown in Algorithm 1. When site j is initialized, H and RQ are empty; all elements of sv are set to zero. After initialization, there are three concurrent threads running in site j:

Thread I is called every time a local operation o is submitted. It executes o locally, adjusts sv by incrementing its j-th element, sets o's timestamp to sv, calls algorithm *LIntegrate* to integrate o into the history H, and broadcasts the transformed o' to the network.

Thread II is called when a remote operation arrives from other sites. It just appends the received operation to RQ.

Thread III first checks whether there are causally-ready operations in *RQ*. The determination is easily done by comparing sv and $o.vt$ [14]. If there are no causally-ready operations in RQ, thread III does nothing. Otherwise, algorithm *RIntegrate* is called to integrate o into H. After

Authorized licensed use limited to: Beijing Jiaotong University. Downloaded on November 11,2022 at 03:32:18 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 *Control Procedure of Site j*

```
1: initialization:
2:         H ← []
3:         RQ ← []
4:         sv ← [0, 0, ..., 0]
   (thread I) submission of local operation o:
5:         execute(o)
6:         sv[j] ← sv[j] + 1
7:         o.vt ← sv
8:         (o', H) ← LIntegrate(o, H)
9:         propagate o'
   (thread II) receipt of operation o from network:
10:        RQ ← RQ · o
   (thread III) invocation of remote operation o:
11:        if exists causally-ready o∈RQ from site k then
12:            (o', H) ← RIntegrate(o, H)
13:            execute(o')
14:            sv[k] ← sv[k] + 1
15:        end if
```

that, the resulting o' produced by *RIntegrate* is executed. Supposing that o is originally submitted at site k, the corresponding element of sv is incremented.

These three threads share variables and must be carefully synchronized in implementation. Thread I is given the highest priority so that local operations are always executed as soon as they are submitted. Thread III is executed only when no local operation is being processed.

A. Algorithm LIntegrate

Procedure *LIntegrate* (*Local-Integrate*) is called by Thread I in the control procedure when a local operation is generated. As specified in Algorithm 2, it mainly achieves the following three goals:

(a) Adding the newly generated operation o into the operation log H. The operation log H records all the executed operations. A newly generated *insert* operation will be added into H_i, while a *delete* operation will be added into H_d.
(b) Keeping the operation log H ordered by the effects relation. As will be shown later, by keeping H ordered by the effects relation, all the involved transformation functions can be done in $O(|H|)$ time.
(c) Excluding all the effects of H_d from o. The reason for this goal is for correctness [8].

When a local operation is generated in the current state, $H_i \mapsto H_d \mapsto o$. By the type of o, there are two cases:

If o is a *delete* operation, to achieve goal (a), o can be appended directly to H_d. To achieve goal (b), however, o needs to be inserted into an appropriate position in H_d. This is done by *mergeDSQ* that adds o into H_d by the effects relation order. Procedure *mergeDSQ* will be presented later in Subsection III-D. To achieve goal (c), a swapping process

Algorithm 2 LIntegrate(o, H):(o', H')

```
1: H'_i ← H_i
2: (o', H'_d) ← swapSQ(H_d, o)
3: if o.type = ins then
4:     H'_i ← mergeISQ(H_i, o')
5: else
6:     H'_d ← mergeDSQ(H_d, o)
7: end if
8: return (o', H'_i · H'_d)
```

(*swapSQ*) is called in line 2 to swap H_d with o. The *swapSQ* procedure is defined in Subsection III-F.

If o is an *insert* operation, because $H_i \mapsto o$ does not hold, o must be transposed to the tail of H_i by swapping it with H_d before adding it into H_i. This swap process is also done by *swapSQ* procedure. After swapping H_d with o, $H_i \mapsto o' \mapsto H'_d$. Now o' can be merged into H_i by calling *mergeISQ* that adds o into H_i by the effects relation order. Procedure *mergeISQ* will be presented in Subsection III-D.

After the above swapping and merging steps, the new history $H'_i \cdot H'_d$ and the resulting o' are returned to the control procedure, in which o' will be propagated to remote sites.

B. Algorithm RIntegrate

Procedure *RIntegrate* (*Remote-Integrate*) is called by Thread III in the control procedure when a remote causally-ready operation is processed. As specified in Algorithm 3, it mainly achieves the following three goals:

(a) Obtaining an appropriate version of o that can be safely executed in the current state.
(b) Adding the remote operation o into the history log H. Similarly to *LIntegrate*, an *insertion* will be added into H_i, while a *deletion* will be added into H_d.
(c) Keeping the log H ordered by the effects relation.

The appropriate version (o') of o that can be safely executed must be defined in the current state. That is, it must satisfy $H_i \cdot H_d \mapsto o'$. By the temporal relations between o and the operations in H, the operations in H can be divided into two classes: operations that happen before o and operations that are concurrent with o. According to Algorithm 2, o does not include the effects of any *delete* operations that happen before it when it is propagated. In addition, let csq_i be the sequence of *insert* operations that are concurrent with o. Then o also does not include the effect of csq_i. Therefore, we must inclusively transform o with csq_i and H_d to obtain the appropriate version defined in the current state. Extracting the concurrent *insert* operation sequence is done by Algorithm 4 which is defined in the next subsection. The inclusive transformations are done by calling *ITOSq* procedures in line 3 and line 4 of Algorithm 3 to include the effects of csq_i and H_d, respectively. As a result, o'' includes the effects of csq_i, and o' includes the effects of both csq_i and H_d.

Authorized licensed use limited to: Beijing Jiaotong University. Downloaded on November 11,2022 at 03:32:18 UTC from IEEE Xplore. Restrictions apply.

Algorithm 3 RIntegrate(o, H):(o', H')

1: $H'_i \leftarrow H_i$
2: $csq_i \leftarrow \text{getConcurrentSQ}(o, H_i)$
3: $o'' \leftarrow \text{ITOSq}(o, csq_i)$
4: $o' \leftarrow \text{ITOSq}(o'', H_d)$
5: **if** o.type = ins **then**
6: $\quad H'_i \leftarrow \text{mergeISQ}(H_i, o'')$
7: $\quad H'_d \leftarrow \text{ITSqO}(H_d, o'')$
8: **else**
9: $\quad H'_d \leftarrow \text{mergeDSQ}(H_d, o')$
10: **end if**
11: return $(o', H'_i \cdot H'_d)$

If o is a *delete* operation, then o' can be safely merged into H_d by calling procedure *mergeDSQ*. This merge achieves both goals (b) and (c), that is, adding o into H_d by the effects relation order.

When o is an *insert* operation, similarly, o'' can be correctly added into H_i by calling procedure *mergeISQ*. However, adding o'' into H_i would invalidate the definition states of the operations in H_d. In order to keep the $H'_i \mapsto H'_d$ relation, the operations in H_d must include the effect of o''. After including the effects of csq_i into o'', $o'' \sqcup H_d$. Therefore the operations in H_d can correct their positions by including the effect of o''. This inclusion is done by the *ITSqO* procedure which is defined in Subsection III-E.

Finally, the updated operation log $H'_i \cdot H'_d$ and the transformed operation o' are returned to the control procedure.

C. Extract Concurrent Operations

In Algorithm 3, we need to extract all operations in H_i that are concurrent with a given remote operation o. Actually most OT algorithms need to do so, e.g., [12], [13], [4], [8]. For given o and sq, in general, they all scan sq left to right: for every $sq[i]$, if $sq[i] \parallel o$, it is appended to sq_c; or if $sq[i] \rightarrow o$, it is swapped with all operations in sq_c and then appended to sq_h. As a result, sq is transposed into $sq_h \cdot sq_c$. This process takes time $O(|sq|^2)$ and is the most time-consuming step in those algorithms [5].

However, this step could be much simplified in ABTSO. Observe that we do not really need the resulting subsequence sq_h and our input sequence is ordered by the effects relation. For example, consider a sequence sq=[o_1, o_2] that is already in effects relation order, where o_1 = ins(0,"ab") and o_2=ins(2,"cd"). Suppose that for the operation o in question we have $o_1 \parallel o$ and $o_2 \rightarrow o$. We can just directly pick up o_1 because swapping o_1 and o_2 changes o_2 but not o_1 and we do not need o_2 at all.

By exploiting this property of effects relation ordering, the procedure of extracting concurrent operations can be drastically simplified, as shown in Algorithm 4. All we need to do is pick up the operations that are concurrent with o one by one. Obviously the time complexity is reduced to linear in the size of the input sequence, i.e., $O(|H_i|)$.

Algorithm 4 $\text{getConcurrentSQ}(o, isq) : csq$

1: $csq \leftarrow [\,]$
2: **for** $(i \leftarrow 0; i < |isq|; i$++$)$ **do**
3: $\quad$ **if** $isq[i] \parallel o$ **then**
4: $\quad\quad csq \leftarrow csq \cdot isq[i]$
5: $\quad$ **end if**
6: **end for**
7: return csq

D. Merge Operation into Sequence

In ABTSO, operations in H are ordered by the effects relation. This property is mainly maintained by the merging procedures, *mergeISQ* and *mergeDSQ*. Every time a new insert or delete operation is merged into H_i or H_d, this property is preserved.

Algorithm 5 $\text{mergeISQ}(sq, o) : sq'$

1: $sq' \leftarrow [\,]$
2: **for** $(i \leftarrow 0;\ i < |sq|$ and $sq[i].pos < o.pos;\ i$++$)$ **do**
3: $\quad sq' \leftarrow sq' \cdot sq[i]$
4: **end for**
5: $sq' \leftarrow sq' \cdot o$
6: **for** $(;i < |sq|;i$++$)$ **do**
7: $\quad sq[i].pos \leftarrow sq[i].pos + |o.str|$
8: $\quad sq' \leftarrow sq' \cdot sq[i]$
9: **end for**
10: return sq'

Procedure *mergeISQ(sq,o)* is specified in Algorithm 5. The precondition is $sq \mapsto o$. After merging, o is part of sq' and sq' is effects equivalent with $sq \cdot o$. Both sq and sq' are in effects relation order. To find the proper position in sq to add o, we need to compare the relative positions of $sq[i]$ and o, where $0 \leq i < |sq|$. Exploiting the operation ordering, we can first swap $sq[i]$ to the end of sq such that $sq[i]' \mapsto o$. Since sq is ordered by effects relation, $sq[i]'$ will be the same as $sq[i]$ for same reasons as explained in Section III-C. Because $sq[i]'$ and o are executed in tandem and $sq[i]' = sq[i]$, we know that $sq[i] \prec o$ if $sq[i].pos < o.pos$, or $o \prec sq[i]$ if $sq[i].pos \geq o.pos$. In the final result sq', the operations that precede o by the effects relation will be kept as-is and the operations that follow o need to be adjusted by $|o.str|$ to account for the effects of o. For example, suppose $sq = [ins(0, \text{“}xy\text{”}), ins(2, \text{“}ab\text{”})]$ and $o = ins(1, \text{“}cd\text{”})$. After the merge, sq' will be $[ins(0, \text{“}xy\text{”}), ins(1, \text{“}cd\text{”}), ins(4, \text{“}ab\text{”})]$.

Similarly, Procedure *mergeDSQ(sq,o)* is specified in Algorithm 6. Note that $|o.str|$ in line 7 is always 1, as defined

Authorized licensed use limited to: Beijing Jiaotong University. Downloaded on November 11,2022 at 03:32:18 UTC from IEEE Xplore. Restrictions apply.

Algorithm 6 mergeDSQ$(sq, o) : sq'$

```
1: sq' ← [ ]
2: for (i ← 0; i < |sq| and sq[i].pos ≤ o.pos; i++) do
3:     sq' ← sq' · sq[i]
4: end for
5: sq' ← sq' · o
6: for (;i < |sq|;i++) do
7:     sq[i].pos ← sq[i].pos − |o.str|
8:     sq' ← sq' · sq[i]
9: end for
10: return sq'
```

in Section II. If $o.str$ is not constrained to be a character, then Algorithm 6 will go wrong since $o.str$ may split [14].

E. IT Algorithms

Now we specify two sequence related inclusion transformation functions, *ITOSq* and *ITSqO*. Function ITOSq(o,sq) transforms an operation o with a sequence sq. Function ITSqO(sq,o) transforms a sequence sq with an operation o. In both cases, the precondition is $sq \sqcup o$.

By the input types, we can define four *ITOSq* functions, that is *ITOSqii*, *ITOSqid*, *ITOSqdi* and *ITOSqdd*. Here we only specify *ITOSqii* because the four functions are similar.

Algorithm 7 ITOSqii$(o, sq) : o'$

```
1: o' ← o
2: for (i ← 0;i < |sq|;i++) do
3:     if (sq[i].pos > o'.pos) then
4:         break
5:     else if (sq[i].pos < o'.pos) then
6:         o'.pos ← o'.pos + |sq[i].str|
7:     else if (sq[i].pos = o'.pos) and (sq[i].id < o.id) then
8:         o'.pos ← o'.pos + |sq[i].str|
9:     end if
10: end for
11: return o'
```

As specified in Algorithm 7, function ITOSqii(o, sq) transforms an insertion o with an insertion sequence sq from left to right to incorporate the effects of insertions in sq one by one. There are three cases to consider:

1) When $sq[i]$.pos $>$ o.pos, it means that o.str will be inserted to the left hand of $sq[i]$.pos and hence we do not need to shift o.pos. By the effects relation ordering, because all operations following $sq[i]$ have effects to the right of $sq[i]$.pos, they will also be skipped.
2) When $sq[i]$.pos $<$ o.pos, it means that o.str will be inserted after $sq[i]$.pos. Hence o.pos should be shifted to the right by $|sq[i].str|$ characters to incorporate the effects of $sq[i]$.

Algorithm 8 ITSqOdi$(sq, o) : sq'$

```
1: sq' ← sq; Δ ← 0
2: for (i ← 0;i < |sq|;i++) do
3:     if (sq[i].pos + Δ) < o.pos then
4:         Δ ← Δ + |sq[i].str|
5:     else
6:         sq'[i].pos ← sq[i].pos + |o.str|
7:     end if
8: end for
9: return sq'
```

3) When o.pos = $sq[i]$.pos, we use a priority scheme, e.g., by comparing their site ids, to break the tie such that the two effects are ordered by their site ids in the result. If $o.id$ is greater, $o.pos$ is shifted to the right.

Next, we specify function *ITSqO*(sq, o). Function *ITSqO* is called in Algorithm 3, and only *ITSqOdi*(sq, o) is needed to be specified, as shown in Algorithm 8. Note that only the operations that follow o in the effects relation need to adjust their positions to incorporate the effects of o. What we need to do is to find these operations in sq. Because sq is ordered by effects relation, there exists some place in sq such that all the operations on its left precede o and all the operations on its right follow o. In order to compare the relative positions between any $sq[i]$ and o, we first swap $sq[i]$ to the head of sq such that $sq[i] \sqcup o$. This can be achieved by excluding effects of all operations that precede $sq[i]$. Suppose the operations $sq[0] \ \ldots sq[i-1]$ delete Δ characters in total, then after the swap $sq[i].pos$ will be $sq[i].pos + \Delta$. By comparing the relative positions (line 3), the effects relations between $sq[i]$ and o can be determined. After that, all operations that follow o just need to adjust their positions by $|o.str|$ to account for o's effects.

F. Swap Algorithms

Function *swapSQ(sq,o)* transposes a sequence sq and an operation o, where $sq \mapsto o$, into sq' and o' such that $o' \mapsto sq'$. By the input types, we can define four functions: *swapSQii*, *swapSQid*, *swapSQdi* and *swapSQdd*. However, since *swapSQ* is only called in Algorithm 2 with H_d as the input, we only need *swapSQdi* and *swapSQdd*. Here we only specify *swapSQdi* because the other works similarly.

Intuitively, function *swapSQdi* can be achieved by swapping every operation in sq with o from right to left. However, by the effects relation, o only needs to shift its position to exclude the effects that precede o, while the operations that follow o need to adjust their positions to include the effect of o. Let Δ be the total number of characters deleted by the operations that precede o. Operation o needs to shift its position by Δ characters to exclude the effects of operations that precede o. The operations that follow o need to shift their positions by $|o.str|$ characters. Similarly to

Authorized licensed use limited to: Beijing Jiaotong University. Downloaded on November 11,2022 at 03:32:18 UTC from IEEE Xplore. Restrictions apply.

Algorithm 9 swapSQdi$(sq, o) : (o', sq')$

1: $o' \leftarrow o$; $sq' \leftarrow sq$; $\Delta \leftarrow 0$
2: **for** $(i \leftarrow 0; i < |sq|$ and $sq[i].pos < o.pos; i$++**)** **do**
3: $\quad \Delta \leftarrow \Delta + |sq[i].str|$
4: **end for**
5: $o'.pos \leftarrow o.pos + \Delta$
6: **for** $(; i < |sq|; i$++**)** **do**
7: $\quad sq'[i].pos \leftarrow sq[i].pos + |o.str|$
8: **end for**
9: return (o', sq')

Algorithm 5, in order to compare the relative positions, we first swap $sq[i]$ to the end of sq. Because sq is ordered by effects relation, $sq[i]$.pos will remain as-is. Consequently, $sq[i].pos$ and $o.pos$ can be directly compared (line 2).

IV. Complexities and Experiments

The space complexity of ABTSO is trivially $O(|H|)$. As specified in Section III, all the procedures only need to scan H once. Hence, the time complexity of ABTSO (i.e., procedure *LIntegrate* and *RIntegrate*) is clearly $O(|H|)$.

In ABTSO, stringwise insertions are processed as-is. However, to avoid deletion splits that complicate transformations, we proactively split every stringwise deletion into characterwise deletions when it is generated by the user. This does not hurt the asymptotic complexity.

According to [5], [14], threads I and III in the control procedure (Algorithm 1) must be mutually exclusive. Consequently, when a remote operation is being integrated by thread III, the user interface is usually locked and thread I must wait. Therefore, the performance of integrating remote operations does impact local responsiveness.

In our experiments, we implemented three algorithms ABT, ABTS, and ABTSO and compare their performance. ABT [8], [6] is our latest characterwise algorithm. ABTS [10] is ABT extended with stringwise operations. We compare the performance of ABT and ABTS to see the impacts of string versus character operations, and compare the performance of ABTS and ABTSO to further see the impacts of our optimization. As discussed in [5], most existing OT algorithms take at least $O(|H|^2)$ to integrate a remote operation. The complexity of ABT is $O(|H_i|^2+|H_d|)$, which is faster than $O(|H|^2)$ by some factor depending on the ratio of insertions in the history. Hence using ABT as the baseline in our experiments does not lose generality.

The algorithms were implemented in C#, compiled using Linux Mono C#, and executed on a HP Compaq nx5000 laptop with a 1.50 GHz Intel Pentium M CPU and 512 MB DDR RAM. The OS is Arch Linux with kernel 2.6.29.

The experiments are designed to study how long it takes to integrate N remote string operations into the local sequence H. Note that N is the size of operation propagation in realtime group editors [7] and hence we assume a relative

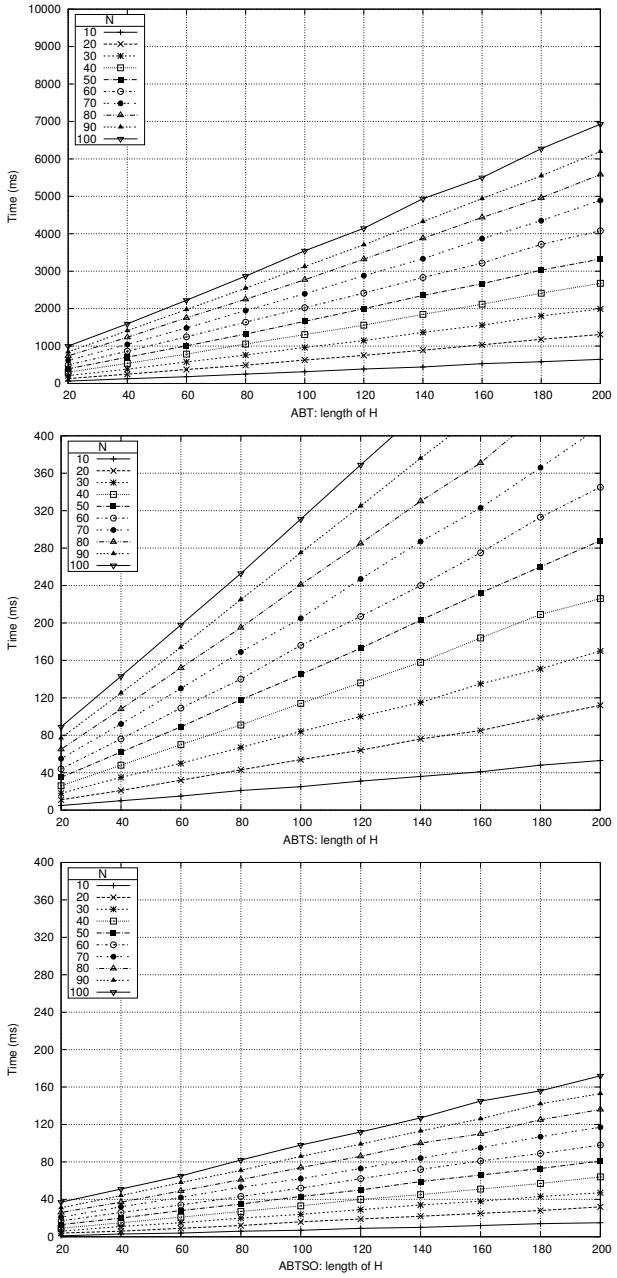

Figure 1. Time to integrate two sequences of string operations on a HP Compaq nx5000 laptop using ABT, ABTS, and ABTSO.

small N. The two sequences are generated concurrently with operation positions roughly uniformly distributed over a large shared document. The document is but an instance of the String type in C#. Each operation is generated with equal probability in any position in the string.

Due to the page limit of this paper, we simplify the experiments such that the insertion ratio is set to 80% and the string length of each operation is 5. Both $|H|$ and N are the number of string operations. $|H|$ varies from 0 to 200 with step 20 and N varies from 0 to 100 with step 10. In

Authorized licensed use limited to: Beijing Jiaotong University. Downloaded on November 11,2022 at 03:32:18 UTC from IEEE Xplore. Restrictions apply.

ABT, every user-input string insertion or deletion is treated as 5 character operations. In ABTS and ABTSO, while string insertions are processed as-is, every delete operation is processed as 5 character deletions. Each experiment is repeated 25 times and the average values are recorded.

The experimental results are as shown in Figure 1. ABTSO is clearly faster than ABTS and ABT by an order of magnitude, while ABTS is faster than ABT by some factor. Since we focus on real-time collaborative applications, we chose small values for the lengths of the sequences so as to observe the effects around the general responsiveness threshold of 100ms [11]. However, their performance would contrast more sharply if longer sequences were used.

It has been well understood in previous work [14], [7] that the granularity of propagation and the frequency of history garbage collection must be chosen such that the 100ms constraint can be ensured. However, they do not address what those values are like in practice. Based on our experimental results, we make the following observations.

First, the data suggests a threshold N for propagating local operations and integrating remote operations. For example, let $|H|$ be 200. ABT takes 645ms to integrate 10 remote operations, implying that N must be far less than 10. ABTS takes 112ms to integrate 20 remote operations, implying that N could go up to between 10 to 20. ABTSO takes 98ms to integrate 60 remote operations, implying that N can be as many as 60. A larger granularity of propagation means better utilization of networking resources since more operations can be multiplexed in one application-level message; and a larger granularity of integration means better awareness between the users since more remote operations can be made visible in a shorter time.

Next, the data suggests a threshold $|H|$ for history garbage collection. For example, let the propagation and integration granularity N be 20. When $|H|$ is 20, ABT takes 130ms to integrate the 20 remote operations, implying that its history size should be no more than 20. When $|H|$ is 180, ABTS takes 99ms to integrate the 20 remote operations, implying that its history size can go up to 180. In contrast, even when $|H|$ is 200, ABTSO only takes 32ms to integrate the 20 remote operations. Hence the $|H|$ threshold for garbage collection in ABTSO can be much larger than 200. These data indicate that under specific responsiveness and granularity requirements, the optimized ABTSO algorithm can considerably mitigate the needs and costs for garbage collection than its unoptimized competitors.

V. Related Work

Traditional consistency control methods such as locking and timestamping either force users to take turns or result in loss of work when concurrent user operations are serialized to achieve convergence. They generally focus more on system efficiency rather than human productivity. In contrast, OT is a lock-free, nonblocking technique that is proposed with the goal to improve individual and group productivity in cooperative work [2]. Several researchers, e.g., [1], [2], [14], have contributed to the understanding of the advantages of OT over these alternative consistency control techniques.

In OT-based systems, it is understood that convergence is not enough. To constrain the converged data content, Sun et al [14] propose to preserve operation intentions. Alternatively, our recent results [8], [6] have established more formal, provable correctness conditions. This work (ABTSO) further optimizes the algorithm performance.

Numerous OT algorithms have been proposed in the context of realtime group editors, e.g., [12], [13], [14], [17]. Most of them need to transpose the entire history to obtain the concurrent operations when integrating a remote operation. This transposition procedure is called *Convert2HC* in [5], which is the most time-consuming step in these algorithms. As analyzed in [5], this procedure is $O(|H|^2)$. In ABTSO, the Convert2HC procedure is substituted by *getConcurrentSQ*, which is only $O(|H|)$ by keeping H ordered in the operation effects relation [8], [6].

To the best of our knowledge, GOT [14] is the first and the only OT algorithm in the literature that has fully presented handling of stringwise operations. The time complexity of GOT is at least $O(|H|^2)$. By comparison, the operation history H in GOT is in the execution order while H in ABTSO is in the effects relation order. ABTSO supports stringwise operations yet simplifies the transformation functions by avoiding deletion splits. As shown in Section IV, this simplification does not undermine efficiency because ABTSO is able to integrate an operation in $O(|H|)$ time.

In real-time group editors, operations are usually propagated and integrated in small batches for resource efficiency. As discussed in [7], they generally propagate operations say every 60 seconds or every 20 operations. However, no previous work to our knowledge has addressed specifically what the impacts of those policies are with regard to the 100ms interactivity requirement, as does our work.

Garbage collection (GC) is to remove from the history of each site operations that have been executed at all sites because they are not needed in future transformations [14]. According to [6], the GC process itself in ABT and hence ABTSO takes $O(|H|^2)$ time in the worst case. By comparison, previous works such as GOT [14] must do GC more frequently due to their $O(|H|^2)$ or higher time complexity when integrating remote operations; ABTSO generally takes more time to do GC yet GC is less frequent due to its $O(|H|)$ time complexity for normal operations.

However, when our work is used in realtime group editors that assume frequent propagation of operations, the system can reach a quiescent state quickly, i.e., all generated operations have been executed at all sites. We can schedule GC in quiescent states such that the entire history can be simply discarded or swapped out to a file without incurring the above expensive $O(|H|^2)$ time transformations.

Authorized licensed use limited to: Beijing Jiaotong University. Downloaded on November 11,2022 at 03:32:18 UTC from IEEE Xplore. Restrictions apply.

VI. Conclusions

This paper presents a novel optimized stringwise transformation algorithm for supporting real-time collaboration over wide-area networks. ABTSO improves the time complexity of the state of art from quadratic to linear. Practically, by transmitting and integrating operations in larger granules, ABTSO is more efficient in terms of communication and computation, which translates into better system performance and usability. As a result, the work can be used in a wide range of parallel and distributed applications that can be abstracted as realtime group editors for lock-free, nonblocking processing at cooperating sites.

To simplify presentation and stay focused on the main contribution, the handling of deletion splits is omitted in ABTSO and will be presented in a journal version of this paper. However, it is worth noting that our unoptimized ABTS algorithm as presented in [10] includes deletion splits. In addition, for space reasons, detailed correctness proofs of ABTSO are also left out because it is an optimization of ABT that has been fully proved [8], [6].

In future research, we will extend this work to support selective undo (e.g., as in [16]). Additionally, transformation-based consistency control algorithms in general need well-designed user interfaces to help end users make sense of the integrated results and detect/resolve conflicts. We also plan to look into these issues in future work.

Acknowledgments

The work is supported in part by the National Natural Science Foundation of China (NSFC) under Grant No. 60736020 and No. 60803118, the National Grand Fundamental Research 973 Program of China under Grant No. 2005CB321905, Shanghai Science & Technology Committee Key Fundamental Research Project under Grant No. 08JC1402700, the Shanghai Leading Academic Discipline Project under Grant No. B114 and the AMD University Cooperation Project.

References

[1] J. Begole, M. B. Rosson, and C. A. Shaffer, "Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems," *ACM Trans. Comput.-Hum. Interact.*, vol. 6, no. 2, pp. 95–132, 1999.

[2] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD'89 Conference on Management of Data*, pages 399–407, Portland Oregon, 1989.

[3] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[4] D. Li and R. Li. Preserving operation effects relation in group editors. In *Proceedings of the ACM CSCW'04 Conference on Computer-Supported Cooperative Work*, pages 457–466, Nov. 2004.

[5] D. Li and R. Li. A performance study of group editing algorithms. In *The 12th International Conference on Parallel and Distributed Systems (ICPADS'06)*, pages 300–307, Minneapolis, MN, July 2006.

[6] D. Li and R. Li. An admissibility-based operational transformation framework for collaborative editing systems. *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, Aug. 2009. Accepted.

[7] D. Li, C. Sun, L. Zhou, and R. R. Muntz. Operation propagation in real-time group editors. *IEEE Multimedia Special Issue on Multimedia Computer Supported Cooperative Work*, 7(4):55–61, 2000.

[8] R. Li and D. Li. Commutativity-based concurrency control in groupware. In *Proceedings of the First IEEE Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'05)*, San Jose, CA, Dec. 2005.

[9] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Survey*, 37(1):42–81, Mar. 2005.

[10] B. Shao, D. Li, and N. Gu. ABTS: A transformation-based consistency control algorithm for wide-area collaborative applications. In *Proceedings of the 5th IEEE Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'09)*, Washington, D.C., Nov. 2009. Invited paper.

[11] B. Shneiderman. Response time and display rate in human performance with computers. *ACM Computing Surveys*, 16(3):265–285, Sept. 1984.

[12] M. Suleiman, M. Cart, and J. Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *IEEE ICDE'98 International Conference on Data Engineering*, pages 36–45, Feb. 1998.

[13] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 59–68, Dec. 1998.

[14] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, Mar. 1998.

[15] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Transactions on Computer-Human Interaction*, 13(4):531–582, Dec. 2006.

[16] D. Sun and C. Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(10):1454–1470, 2009.

[17] N. Vidot, M. Cart, J. Ferrie, and M. Suleiman. Copies convergence in a distributed realtime collaborative environment. In *Proceedings of ACM CSCW'00 Conference on Computer-Supported Cooperative Work*, pages 171–180, Dec. 2000.

Authorized licensed use limited to: Beijing Jiaotong University. Downloaded on November 11,2022 at 03:32:18 UTC from IEEE Xplore. Restrictions apply.