A Fast Operational Transformation Algorithm for Mobile and Asynchronous Collaboration

Bin Shao, *Member*, *IEEE*, Du Li, *Member*, *IEEE*, and Ning Gu, *Member*, *IEEE*

Abstract—In a mobile collaboration environment, the shared files are often replicated so that the users can work in parallel during periods of disconnection. When reconnected, sequences of updates made by different users are merged to produce a consistent view of shared files. This paper presents a novel transformation-based merging algorithm for supporting mobile collaboration. Compared to alternative optimistic consistency control methods, it can merge any updates to the same file automatically at the finest granularity without causing loss of work. Moreover, it improves the time complexity of the state-of-the-art transformation-based merging algorithms from $O(n^3)$ to $O(n)$, where n is the size of either sequence when their sizes are comparable. This improvement is drastic and important especially for mobile devices that run on batteries and have limited processing power.

Index Terms—Collaboration, concurrency control, data consistency, mobile computing, operational transformation, optimistic replication.

1 Introduction

Mobile computing devices such as laptops, netbooks, PDAs, tablets, and cellular phones are becoming more and more pervasive. Compared to their desktop counterparts, mobile devices typically operate on batteries and wireless networks for mobility and their connections are usually intermittent. These characteristics pose significant challenges on the design of collaborative systems that support collaboration in a mobile environment.

One of the major challenges is data consistency. For performance reasons such as responsiveness and availability, the shared data are often replicated on mobile devices. As a result, the users are able to work on their local data replicas in disconnected mode and synchronize with their collaborators when the devices are reconnected and the users feel "ready."

In the presence of intermittent connection, consistency control has to be optimistic [2]: Pessimistic protocols such as two-phase locking (2PL) are often too expensive for the system and counterproductive for the users since they have to wait for locks. In contrast, optimistic protocols allow data replicas to diverge so that users can still make progress concurrently even when disconnected; consistency is eventually achieved when concurrent updates are merged and conflicts resolved.

A number of optimistic replication techniques have been proposed in the literature, to name but a few, [3], [4], [5], [6]. In general, these techniques consider two primitive operations, read and write; updates to the same object are usually executed in the same total order to ensure convergence in different data replicas. As a consequence, however, they may suffer from loss of updates since only one write can eventually "win" among multiple concurrent writes that conflict, i.e., target the same object. Therefore, those techniques are usually used in applications in which concurrent updates rarely conflict, e.g., distributed file systems [3], [5].

Operational transformation (OT) [7], [8] is an optimistic technique that is originally proposed in the context of collaborative document editing. In this domain, the users often need to edit the same document simultaneously, in which blocking, coarse-grained consistency control is often unacceptable [9]. OT is more suitable due to its lock-free, nonblocking properties. In OT, data updates are usually represented in two primitives, insert and delete; any pair of updates can be transformed to commute; consistency can be achieved without enforcing a total order of execution. At one extreme, OT can automatically merge all updates to the same file, preserving all their effects for deferred conflict resolution. The technique can handle both unstructured and structured contents, including text files, source code, Web pages and XML documents [8], [10], [11].

We study how to extend OT techniques to mobile systems in this paper. Due to the needs to support disconnected operation, they must be able to efficiently merge update sequences of arbitrary lengths. However, most existing OT algorithms are designed for supporting real-time group editing in which updates are frequently propagated and merged. As a result, they focus on how to transform one operation at a time rather than how to merge long sequences. As will be shown in Section 2, the state-of-the-art OT algorithm takes $O(n^2)$ to merge one remote update and $O(n^3)$ to merge a long sequence, where n is the size of the local operation log. Those algorithms would be suboptimal on mobile devices.

This paper presents a novel OT-based algorithm optimized for supporting mobile and asynchronous collaboration. As the main contribution, we propose an admissibility-based sequence transformation (or ABST) algorithm, which

- *B. Shao and N. Gu are with the School of Computer Science, Fudan University, Shanghai 200433, China. E-mail: {binshao, ninggu}@fudan.edu.cn.*
- *D. Li is with Nokia Research Center, 955 Page Mill Road, Palo Alto, CA 94304. E-mail: lidu008@gmail.com.*

Manuscript received 30 Apr. 2009; revised 4 Sept. 2009; accepted 15 Sept. 2009; published online 31 Mar. 2010.
Recommended for acceptance by P. Stenstrom.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2009-04-0190.
Digital Object Identifier no. 10.1109/TPDS.2010.64.

improves the time complexity of the state of the art from $O(n^3)$ to $O(n)$ when merging two sequences, where n is the size of either sequence when their sizes are comparable. Our experiments show a remarkable performance improvement: For example, it takes *59 minutes* in a recent OT algorithm [12] versus *1.5 seconds* in this work on a mobile device to integrate two sequences each of 3,000 operations.

This efficiency improvement is achieved primarily by maintaining a special order on operations in each of the involved sequences based on the so-called operation effects relation [12]. As a result, by exploiting this operation order, the key steps in merging any two concurrent sequences can all be achieved in linear time. In this paper, we will also prove correctness of the ABST algorithm. In fact, the algorithm is so named because it is based on the admissibility theory [12], [13] that formalizes correctness conditions to address some chronic correctness problems in previous work. In other words, the efficiency improvement in this work is well grounded on correctness of the algorithm.

The rest of this paper is organized as follows: First, in Section 2, we analyze and compare related works to establish the context for this research. After that, Section 3 introduces the basic concepts and notations to be used in this paper. Sections 4 and 5 present the ABST algorithm. Section 6 gives an example to further illustrate its working. Sections 7 and 8 analyze its correctness and performance, respectively. Then Section 9 discusses conflict detection. Finally, Section 10 summarizes the contributions and future directions.

2 Context: Related Works

This work mainly involves the following three aspects: data and file synchronization, optimistic replication, and OT. We compare related works regarding these aspects accordingly.

2.1 Data and File Synchronization

Data synchronizers such as ActiveSync and IntelliSync focus on synchronization of structured data such as contacts and calendars, especially between a desktop computer and a mobile device. Since the data has fixed structures and concurrency is limited, conflicts can be easily detected and resolved by predefined policies that are usually application specific.

There has been significant research on file synchronizers, e.g., [14], [15]. In general, they detect conflicts at the file level. When two updates to the same file are concurrent, a conflict is reported for the user to manually merge the updates. Updates to the same file are not merged automatically. In comparison, this work focuses on fine-grained file content synchronization.

Tools such as Rsync can merge updates to unstructured files at a finer granularity than the file level. It chops a file into fixed-size, nonoverlapping chunks and computes their checksums. Then checksums of different replicas are compared. Chunks that do not match are transmitted from the sender to the recipient and merged to make the two replicas identical. Some of the chunks may be overwritten. The main limitations are its unidirectional synchronization and chunk-based granularity.

Version control systems such as RCS, CVS, SVN and GIT are widely used in software projects for coordinating and synchronizing parallel modifications to source code. In general, they work at the granularity of lines. Two concurrent changes are conflicts if they happen to the same line, and it is usually considered that only the user can resolve the conflicts. Manual merging is required even if the changes do not really conflict. Those systems are asynchronous because the users usually synchronize at the granularity of block updates instead of single-character edits. It is possible to integrate OT-based techniques such as ABST in those systems to help reduce manual work by implementing more fine-grained merging without losing efficiency. In general, the idea of OT is to preserve the effects of all updates and assume the "right" user interfaces and mechanisms for the human users to detect and resolve semantic conflicts, e.g., grammatical errors [10], [16]. Merging can be automated as long as concurrent updates do not really cause semantic conflicts.

2.2 Optimistic Replication

Optimistic replication techniques are widely used in mobile systems, such as Coda [3], [17], Bayou [4], Rumor [5], and IceCube [6]. Among them, Coda and Rumor are distributed file systems which assume concurrent updates to the same file are rare; hence, they generally consider those updates as conflicts that are resolved by human users.

Bayou [4] guarantees consistency by enforcing that all sites execute the same set of writes in the same total order. By comparison, our work achieves consistency without enforcing a totally ordered execution path. In addition, Bayou has an abstract concept of *session* that bears some resemblance to our concept of *transactional sequence*. A session is a sequence of read/write operations that need to be consistent in effects, while operations in a transactional sequence are required to be executed without interruption at local and remote sites.

In IceCube [6], consistency is achieved by replaying merged operation logs from the same initial state; operations in the logs are reordered to minimize conflicts; when conflicts occur, the system is blocked and the resolution is delegated to the user. By comparison, our approach is also log-based; operations in the log are kept separately by their types; when a sequence is integrated, operations are reordered to optimize the algorithm complexity rather than avoid conflicts.

2.3 Operational Transformation

The approach taken in this work is an orthogonal optimistic replication technique called operational transformation (OT). Among the most differentiating properties of OT, positions of objects in the data structure (e.g., a linear array) are used for uniquely identifying objects, which is much cheaper than using global object ids; two primitives, insert and delete, are used in place of write and any pair of updates can be transformed to commute, which avoids overwriting and eliminates the need for the same total order of execution to achieve consistency. These properties make it possible to support fine-grained updates, merging, and conflicts handling.

2.3.1 General Issues in Previous Work

A plethora of OT algorithms has been proposed over the past decade, e.g., [8], [10], [18], [19], [20]. We survey them along the following two dimensions.

First, most OT algorithms are based on the framework of Sun et al. [10], which includes three conditions, namely, convergence, causality, and intention preservation. The notion of intention preservation is intuitive and has been adopted widely in a range of collaborative editing systems. Nevertheless, due to its generality, intention preservation is not intended to be a formal condition for purposes of correctness proofs in specific applications, e.g., group text editors. Some of the remaining challenges have been confirmed, e.g., as in [13], [16], [21], [22].

Second, most OT algorithms, e.g., [8], [10], [12], [16], [18], [21], are designed for real-time group editors which assume frequent propagation of operations. Under this assumption, the local history (H) is usually kept short enough to warrant efficiency. Hence, although operations are usually propagated in batches [23], those algorithms only address how to integrate one remote operation at a time. On the other hand, they keep operations in the history H in the order of execution. Hence, every time a remote operation o is integrated, they generally need to transpose the history such that those operations that happened before o precede those that are concurrent with o; then o is transformed with those concurrent operations [8]. The transposition step alone takes $O(|H|^2)$ time [24]. While they may not have performance problems in real-time group editors, those algorithms deserve further optimizations for mobile computing due to the need to support disconnected operation.

Shen and Sun [25] proposed one of the few OT algorithms for asynchronous collaboration. Their work is based on the notion of intention preservation. In addition, it is tailored for a client/server architecture and achieves convergence by making use of the server. It orders operations in sequences in their execution order, as in other OT works. When transforming two sequences, it takes time quadratic in the size of the two sequences. Nevertheless, their work is complementary to ours: Their proposal to compress the operation sequences [26] could be leveraged in our future research to reduce the size of sequences for even better efficiency; and our work orders input sequences by the effects relation to achieve linear merging time, which could be leveraged in their work.

Prior work in [27] also involves how to merge two sequences (edit scripts obtained by diffing [28]) in linear time. However, its focus is on how to turn single-user editors into real-time group editors without modifying source code. Hence, its algorithm is simplified to work only for two sites such that every time two sequences are transformed they must be defined in the same state. This requires that the two users sync at the same time, which is a severe constraint in mobile environments. Nevertheless, results in this work can be leveraged in the system of [27] for improved flexibility in sync.

2.3.2 The State of the Art and Comparisons

Admissibility-Based Transformation (ABT) [12], [13] is our latest theoretical framework that proposes two formal conditions, namely, causality and admissibility preservation. For correctness reasons, ABT maintains history H as the concatenation of an insertion subsequence (H_i) and a deletion subsequence (H_d). Operations in either subsequence are in execution order. As analyzed in [24], ABT takes $O(|H_i|^2 + |H_d|)$ time to integrate a remote operation because only H_i is transposed.

In our performance experiments (Section 8), we will use ABT as the baseline for comparisons, mainly for two reasons: First, although there exists alternative understanding of OT correctness [10], by our understanding, we prefer to compare performance of ABST to those that are also formally proved and fully presented for ease of validation. Second, ABT is, in general, more efficient than other OT algorithms to our knowledge. Among the most established works, SOCT2 takes $O(|H|^2)$ time to integrate one remote operation, which is slower than ABT by some factor because H_i is a subsequence of H [24]; and according to [20], the complexity of GOTO [8] is exponential in the size of the history H.

An algorithm that takes $O(|H|^2)$ or more time to integrate one remote operation would be suboptimal for mobile computing. Suppose that when a sequence T is integrated, $|T| = m$ and $|H| = n$. Each time one operation from T is integrated, it is added to H, increasing its size by one. Hence, using an $O(|H|^2)$ algorithm, the time to integrate T is $O(\sum_{k=n}^{n+m-1} k^2) = O(n^2 \cdot m + n \cdot m^2 + m^3)$. When m and n are comparable, it is $O(|H|^3)$.

ABST is follow-up work of ABT and its correctness can be formally proved (Section 7). The notion of *effects relation* in ABT is leveraged to order operations by the relative position of their effects, i.e., the objects they insert and delete. The main differences are that: 1) ABST propagates and integrates operations in transactions, and 2) ABST keeps histories and transactions ordered by the effects relation. As a result, by exploiting this ordering property, time complexities of the four key functions in ABST, getConcurrentSQ, ITSQ, mergeSQ, and swapSQ, are linear. Also, note that these functions are for handling transactional sequences, and hence, all new in ABST.

3 Background: Concepts and Notations

We first illustrate the basic ideas behind OT, define the data model and primitives, and explain the key concepts, effects relation and admissibility. We also introduce some useful notations, which are summarized in Table 1.

3.1 Basic Ideas of OT

The basic ideas of OT can be illustrated as shown in Fig. 1. Suppose that a file containing string "ac" is replicated at two sites. Site 1 performs operation $o_1 = \mathrm{ins}(1, \mathrm{b})$ locally which inserts "b" at position 1 to change the content to "abc." In parallel, site 2 performs $o_2 = \mathrm{del}(0)$ locally to delete the object "a" at position 0 to change the content to "c." These two operations are propagated to all sites.

At sync time, site 1 needs to merge remote operation o_2 with local operation o_1. By inspecting their positions, o_1 inserts on the right side of where o_2 deletes. Hence, the position of o_2 is not affected and can be executed as-is in state "abc," which results in content "bc." In this case, the execution form of o_2 is $o_2' = \mathrm{del}(0)$.

On the other hand, at site 2, if o_1 were executed as-is in current state "c," the wrong content "cb" would be yielded. This is because the execution of o_2 on the left of where o_1 inserts has invalidated the position of o_1. Hence, o_1 needs to

TABLE 1
Main Notations and Their Semantics: o and o_i Denote Operations, and sq and sq_j Denote Sequences

Notation	Brief Description
o.id	the id of site that originally generates o
o.v	the timestamp of o when it is generated
o.type	the operation type of o, either ins or del
o.pos	the position of o relative to the data model
o.c	the effect of o, i.e., the object o inserts or deletes
$o_1 \prec o_2$	effect o_1.c appears on the left of o_2.c in some state
$o_1 \rightarrow o_2$	o_1 happens before o_2
$o_1 \parallel o_2$	o_1 and o_2 are concurrent
$o_1 \sqcup o_2$	o_1 and o_2 are contextual equivalent w.r.t. same state
$o_1 \mapsto o_2$	o_1 and o_2 are contextually serialized
$[o_1, o_2]$	a 2-operation sequence in which $o_1 \mapsto o_2$
$\|sq\|$	the number of operations in sequence sq
$sq[i]$	the operation in position i of sq where $0 \leq i < \|sq\|$
$sq_1 \cdot sq_2$	a concatenated sequence where $sq_1 \mapsto sq_2$
$sq \cdot o$	a concatenated sequence where $sq \mapsto o$

be transformed (i.e., by shifting its position) into form $o_1' = \mathrm{ins}(0, \mathrm{b})$ such that it can be correctly executed in current state of site 2. As a result, site 2's content becomes "bc" and the two replicas converge despite different orders of execution.

In traditional distributed systems, the same total order of operations is often enforced at all sites such that the replicas converge in the same state. Obviously, different total orders usually yield different results. In Fig. 1, for instance, if the total order is that o_1 precedes o_2, site 2 must first undo o_2, and then, execute o_1 and o_2 in order; if the total order is o_2 precedes o_1, on the other hand, site 1 must first undo o_1, and then, execute o_2 followed by o_1. Although the former total order produces the right result "bc," the latter yields the wrong result "cb" that violates the intention of o_1 to insert "b" before "c" [10].

Traditional systems generally focus on convergence and do not address which total order is "right." In contrast, OT-based systems further require that the execution of operations preserves operation intentions [10] or admissibility [12], [13]. Admissibility differs mainly in that it is a formal and provable condition.

3.2 Model of Data and Primitive Operations

To simplify discussions, we model the shared data as a linear string (or sequence) and only consider two primitive operations, ins(p, c) and del(p), which insert and delete one object (character) at position p, respectively. For any operation o, we define the following five attributes: $o.id$ is the id of the site that originally generates it; $o.v$ is the time stamp when o is generated; $o.type$ is either ins or del; $o.pos$ is the target position; and $o.c$ is the **effect object** that o inserts or deletes.

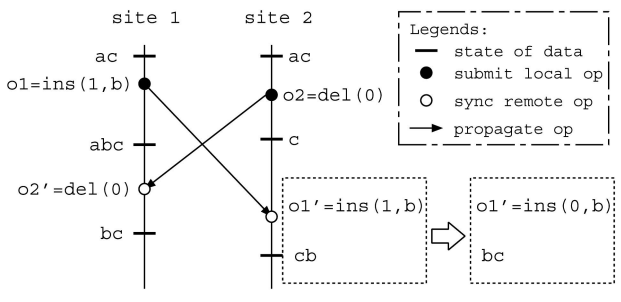


Fig. 1. Concurrent updates are transformed to commute at sync time without overwriting each other.

Note that $o.pos$ is an integer since the shared data are modeled as a linear string. According to [11], we can denote $o.pos$ as a vector of integers to model a tree structure, with little modification to the algorithms. Further, according to [19], even operations in sophisticated office and design software can be mapped to a linear or tree-like data model. Theoretically, an update operation that modifies an attribute of an object could be denoted as a deletion of the original object followed by an insertion of the object with the new attribute. Therefore, our model does not lose generality although simplified.

3.3 Effects Relation and Admissibility

Here, we only conceptually explain the concepts. A more rigorous treatment has been presented in [12], [13].

Notation $\prec$ denotes the relation between objects. In the scenario of Fig. 1, for example, due to the initial state "ac," we have $a \prec c$. After "b" is inserted, the relation is extended to $a \prec b \prec c$. In any state of the shared data that ever appears in our system, which is a linear structure in our model, the relation $\prec$ is a total order.

We extend $\prec$ to denote the **effects relation** between operations, which is defined as the position order between the effects of two operations. In the above scenario, since the effect object o_2 deletes ("a") appears on the left of the effect object o_1 inserts ("b") in some state "abc," we denote their effects relation as $o_2 \prec o_1$.

When any operation o is generated, the relation between its effect object $o.c$ and the objects in its generation state is determined: If o is an insertion, it creates a new object $o.c$ and introduces new relationships between $o.c$ and existing objects; if o is a deletion, it removes an existing object $o.c$ but does not change the relationships between $o.c$ and other existing objects.

The invocation of any operation locally in its generation state is not interfered by concurrent operations. However, when a remote operation o is invoked, we must transform o into a form o' such that o' is **admissible** in its execution state. That is, the execution of o' should not introduce new relationships that contradict the relation $\prec$ that has been

established, e.g., rendering it cyclic. For example, in the above scenario, if o_1 were invoked at site 2 in the form of ins(1,b), it would lead to relation $c \prec b$, which violates the relation $b \prec c$ that has been established when o_1 is generated at site 1.

Note that the effects relation $\prec$ is a **partial order**, e.g., when two concurrent operations insert at the same position. In Fig. 1, suppose that o_3 inserts "d" also between "a" and "c" and $o_1 \parallel o_3$. When o_1 and o_3 are generated, $a \prec b \prec c$ and $a \prec d \prec c$, and we cannot determine their effects relation. However, whenever they are transformed with each other at some site, to be admissible, their effects relation, either $o_1 \prec o_3$ or $o_3 \prec o_1$, must be unambiguously determined. As a result, relation $\prec$ eventually becomes a **total order** in the system [12], [13].

3.4 More Notations on Operations and Sequences

In scenarios like the above, concurrent execution of operations can easily lead to inconsistencies. Hence, when merging two operations, we need to determine their temporal relationship, e.g., whether or not they are concurrent. As is conventional in distributed systems [7], [29], we use vector time stamps for determining the happens-before ($\rightarrow$) and concurrent ($\parallel$) relations. In the above scenario, we have relation $o_1 \parallel o_2$.

In addition, note that $o.pos$ is always relative to a state of the document. Borrowing established notations from [8], if two operations are defined in the same state, they are said to be contextually equivalent ($\sqcup$); if one is defined in the state resulted from executing another operation, they are said to be contextually serialized ($\mapsto$). In Fig. 1, we have $o_1 \sqcup o_2$, $o_1 \mapsto o_2'$, and $o_2 \mapsto o_1'$.

An operation sequence $sq = [o_1, o_2, \ldots, o_n]$ means that all operations in sq are contextually serialized. That is, $o_1 \mapsto o_2 \cdots \mapsto o_n$. The number of operations in sq is denoted as $|sq|$. Notation $sq[i]$ refers to the operation in position i of sq, where $0 \leq i < |sq|$. If two sequences sq_1 and sq_2 are contextual serialized, denoted as $sq_1 \mapsto sq_2$, we can concatenate them into a new sequence $sq = sq_1 \cdot sq_2$. Generalizing notations $\mapsto$ and $\cdot$, for an operation o and a sequence sq, if $sq \mapsto o$, we can append o to sq obtaining a new sequence $sq' = sq \cdot o$.

4 Sequence Transformation Algorithm

A history buffer H is maintained at each site which records all operations that have been executed at that site. ABST propagates and integrates a sequence of operations at a time as if it is a "transaction." By keeping operations in the effects relation order, integrating a remote sequence can be achieved in linear time.

We present the ABST algorithm in Sections 4 and 5. Section 4.1 first introduces the concept of "transactional sequence." Section 4.2 discusses how a remote sequence T is integrated with local history H. The rest of Section 4 elaborates all the functions that are called in the integration algorithm. Then, Section 5 discusses how to obtain a local transactional sequence before it is propagated.

For the scope of this paper, we assume that all sites start from the same version of the shared data and all sequences are eventually received by all sites in the system. That is, we do not address exactly how the shared data are replicated and how the sequences are propagated. Those problems and their solutions are already well understood in the literature [2].

4.1 Transactional Sequence

A human user is usually not interested in propagating or receiving single ins/del operations, but in a larger editing session, applied as a unit, somewhat akin to a database transaction [15]. We call it a *transactional sequence*, which conceptually has two aspects:

On the one hand, the user performs a batch of editing operations locally first, and then, propagates them to remote sites when she feels "ready" for sharing. During a local transaction, no remote operation is merged so as not to distract the user from her own work. This way, a sequence T of operations is accumulated and propagated in its entirety to remote sites.

On the other hand, at sync time, a number of sequences could have been received from remote sites. These sequences are kept in a receiving queue RQ. When the user feels "ready" for merging, each time a sequence T from RQ is integrated into H after transformation. The transformed sequence T' is then applied to the local data replica in its entirety.

A transactional sequence has the following properties: All operations in a given transactional sequence T have the same site id; their vector time stamps are equivalent in all corresponding elements except for the one representing that site. Hence, we can use $T.id$ for the site id, and $T.type$ for the type if all operations in T are of the same type. It can be shown that for any operation o that is not in T, if o is concurrent with any $T[i]$, then o must be concurrent with any other $T[j]$, where $0 \leq i, j < |T|$. Similarly, for any operation o' that is not in T, if it happens before any $T[i]$, it must also happen before any other $T[j]$. For convenience, we denote the relations as $o \parallel T$ and $o' \rightarrow T$, respectively.

4.2 Integrate One Remote Sequence

In our work, we maintain the history H at each site as $H_i \cdot H_d$, where H_i is a sequence of all insertions and H_d is a sequence of all deletions in H. Furthermore, operations in each of these two subsequences are ordered by the effects relation $\prec$. Similarly, a transactional sequence T is maintained as $T_i \cdot T_d$, where T_i and T_d are sequences of insertions and deletions that are ordered by the effects relation, respectively. History H is initially empty and grown incrementally as local and remote transactions are integrated. As explained in Section 2.3 and will be elaborated in Sections 7 and 8, sequences H and T are so ordered for correctness and efficiency reasons.

Let H^k be site k's history. When a sequence T is generated at site x, $H^x \rightarrow T$ must hold. When T is integrated at site y, it must be *causally-ready* at site y, i.e., all operations that happen before T have been executed at site y and included in H^y. Before $T = T_i \cdot T_d$ is propagated from site x, it must have been transformed with H^x such that $H_i^x \mapsto T \mapsto H_d^x$. That is, T does not include the effects of any deletions that happen before T. We will explain how to obtain T that satisfies this condition in Section 5 and why in Section 7.

Algorithm 1 shows how a remote sequence T is integrated with local history H. Properties of the two input sequences are as explained above. In the output, T' is the result of transforming T with H; H' already includes all operations in T; and the same ordering properties are maintained in H' and T'. After function Integrate returns, the resulting T' is ready to be executed in the current state of the local data replica.

Algorithm 1. Integrate(T, H): (T', H')
1: $csq_i \leftarrow \text{getConcurrentSQ}(T_i, H_i)$
2: $T_i'' \leftarrow \text{ITSQ}(T_i, csq_i)$
3: $T_i' \leftarrow \text{ITSQ}(T_i'', H_d)$
4: $H_i' \leftarrow \text{mergeSQ}(H_i, T_i'')$
5: $H_d'' \leftarrow \text{ITSQ}(H_d, T_i'')$
6: $csq_d \leftarrow \text{getConcurrentSQ}(T_d, H_i')$
7: $T_d'' \leftarrow \text{ITSQ}(T_d, csq_d)$
8: $T_d' \leftarrow \text{ITSQ}(T_d'', H_d'')$
9: $H_d' \leftarrow \text{mergeSQ}(H_d'', T_d')$
10: return $(T_i' \cdot T_d', H_i' \cdot H_d')$

The essential idea of integrating a remote operation o is to incorporate the effects of operations that have been executed locally (already in H) but not yet included in o. The execution of those operations has led to current state of the data but may have invalidated the position of o relative to the current state. In the literature [8], the process of incorporating effects of one operation into another is called inclusion transformation or IT.

The same idea applies when integrating a remote sequence T. As explained in Section 4.1, when Algorithm 1 is called, T is causally ready. That is, any operation in H must either happen before T or be concurrent with T. In addition, T has excluded the effects of all deletions that happen before T in its originating site. We know that H is maintained as $H_i \cdot H_d$, or an insertion sequence H_i concatenated by a deletion sequence H_d. Hence, T includes effects of some of the operations in H_i (that happened before T) but none of the operations in H_d. By the spirit of OT, we need to incorporate the effects of all insert operations in H_i that are concurrent with T and all delete operations in H_d.

Accordingly, Algorithm 1 consists of two parts: lines 1-5 integrate T_i and lines 6-9 integrate T_d. We explain these two parts as follows:

First, to integrate T_i, we need to extract all operations in H_i that are concurrent with T_i, resulting in sequence csq_i. This is achieved by function getConcurrentSQ. Next, we call function ITSQ to incorporate the effects of csq_i into T_i, resulting in T_i''. Then, we incorporate the effects of H_d into T_i'', yielding T_i'. After that, we call function mergeSQ to merge T_i'' into H_i, yielding H_i'. Now all insertions in T are integrated into H.

Note, however, that the last step to merge T_i'' into H_i invalidates the definition state of H_d. This is because H_d is defined in the state obtained by executing H_i, i.e., $H_i \mapsto H_d$. After the merging, property, $H_i' \mapsto H_d$ no longer holds due to the effects of T_i''. Hence, we need to transform H_d by calling function ITSQ to incorporate the effects of T_i'', yielding H_d''. Now $H_i' \mapsto H_d''$ holds.

Second, to integrate T_d, we first extract all operations in H_i' that are concurrent with T_d, resulting in csq_d. Note that csq_d and csq_i have exactly the same set of operations. However, as a result of merging T_i in step 4, operations in csq_d already include the effects of T_i, and hence, the two sequences T_d and csq_d are relative to the same definition state, that is, $T_d \sqcup csq_d$. Now calling function ITSQ, we first incorporate effects of csq_d into T_d, yielding T_d'', and then, incorporate effects of H_d'' into T_d'', yielding T_d'. After that, we call mergeSQ to merge T_d' into H_d'', yielding H_d'.

When Algorithm 1 returns, we have fulfilled the first goal of transforming T with H such that $T' = T_i' \cdot T_d'$ can be executed in current state, and the second goal of merging T such that H is updated to $H' = H_i' \cdot H_d'$. In the rest of this section, we will explain the three functions, namely, getConcurrentSQ, ITSQ, and mergeSQ.

4.3 Extract Concurrent Subsequence

Algorithm 2 extracts all operations that are concurrent with sequence T from sequence isq. Input T is a transactional sequence and input isq is an insertion sequence that is ordered by the effects relation. The output csq is a sequence in which all operations are concurrent with T and ordered by the effects relation.

Algorithm 2. $\text{getConcurrentSQ}(T, isq) : csq$
1: $csq \leftarrow [\,]$
2: **for** $(j \leftarrow 0; j < |isq|; j++)$ **do**
3: **if** $(isq[j] \parallel T)$ **then**
4: $csq = csq \cdot isq[j]$
5: **end if**
6: **end for**
7: return csq

By discussions in Section 4.2, any operation in isq either happens before T or is concurrent with T. For example, without loss of generality, let it be $[h_1, c_1, h_2, c_2, h_3]$, where h_1, h_2, h_3 happen before T and c_1, c_2 are concurrent with T. Conceptually, we have to transpose these operations such that the result $isq' = [h_1', h_2', h_3', c_1', c_2']$. That is, $[h_1', h_2', h_3'] \mapsto [c_1', c_2']$, meaning that all those that happen before T are contextually serialized before all those that are concurrent with T. To achieve so, we need to "swap" every h_i with every c_j on its left.

As in [8], a transposition algorithm could be devised as follows: let csq hold the partial results; scan isq from left to right, for every $isq[j]$, if $isq[j] \parallel T$, append it to csq; or if $isq[j] \rightarrow T$, swap it with every operation in csq right to left. The time complexity is $O(|isq|^2)$.

However, this process can be greatly simplified if we consider the property of input isq that all operations are insertions and ordered by the effects relation. By this property, for any $o_i = isq[i]$, we have the effects relation $o_i \prec o_{i+1}$, where $0 \le i < i+1 < |isq|$. Swapping o_i and o_{i+1} means that in an alternative execution order, o_{i+1} is executed before o_i. Swapping execution order of two operations should not change the effects relation. For example, consider state "abc." After executing sequence [ins(1,x), ins(3,y)], the state becomes "axbyc." Transposing these two operations results in sequence [ins(2,y), ins(1,x)], which applied on "abc" yields the same state "axbyc." More generally, let $[o_{i+1}', o_i']$ be the result of swapping $[o_i, o_{i+1}]$. The relation between their positions must be $o_i'.\text{pos} = o_i.\text{pos}$ and $o_{i+1}'.\text{pos} = o_{i+1}.\text{pos} - 1$. That is, the swapping changes $o_{i+1}.\text{pos}$ but not $o_i.\text{pos}$.

This property leads to the following simplification of algorithm: each time an operation $isq[j] \rightarrow T$ is swapped with the partial result csq, the swapping does not affect the positions of operations in csq. Also, note that we do not really need those operations in isq that happen before T. Therefore, we just need to directly pick up those in isq that are concurrent with T. This is exactly how Algorithm 2 is designed. Obviously, its time complexity is $O(|isq|)$.

4.4 Transform Two Sequences

Function ITSQ(sq_1, sq_2) inclusively transforms sequence sq_1 with sq_2, i.e., to incorporate the effects of sq_2 into sq_1. The precondition is $sq_1 \sqcup sq_2$, meaning that the two input sequences are defined in the same state. The postcondition is $sq_2 \mapsto sq_1'$, meaning that the output sequence sq_1' is as if it were to be executed after sq_2.

By operation types of the two input sequences, we can define four ITSQ functions. We first define function ITSQii for transforming two insertion sequences, as in Algorithm 3. The other three, ITSQid, ITSQdi, and ITSQdd, can be defined similarly. In principle, we need to transform every $sq_1[j]$ with every $sq_2[i]$. However, some of the transformations can be saved because the two input sequences are ordered by the effects relation.

Algorithm 3. ITSQii(sq_1, sq_2) : sq_1'

```
 1: i ← 0; j ← 0; sq1' ← sq1
 2: while (i < |sq2|) and (j < |sq1|) do
 3:   ipos ← sq2[i].pos − i
 4:   jpos ← sq1[j].pos − j
 5:   if ipos < jpos then
 6:     i ← i + 1
 7:   else if (ipos = jpos) and (sq2[i].id < sq1[j].id) then
 8:     i ← i + 1
 9:   else
10:     sq1'[j].pos ← sq1'[j].pos + i
11:     j ← j + 1
12:   end if
13: end while
14: for (; j < |sq1|; j++) do
15:     sq1'[j].pos ← sq1'[j].pos + i
16: end for
17: return sq1'
```

For any two given operations $sq_1[j]$ and $sq_2[i]$, we can directly transform $sq_1[j]$ with $sq_2[i]$ only when they are defined in the same state and their positions can be compared. Consider these two sequences are defined in the same state and operations in these two sequences are contextually serialized. To bring $sq_1[j]$ and $sq_2[i]$ to the same state, we only need to swap $sq_1[j]$ to the front of sq_1 and also swap $sq_2[i]$ to the front of sq_2. By how the operations are ordered in sq_1, we know that all operations ordered on the left of $sq_1[j]$ have effect objects that precede the effect object of $sq_1[j]$. So, the swapping results in operation $sq_1[j]'$ such that $sq_1[j]'.\text{pos} = sq_1[j].\text{pos} - j$ to account for the j objects on its left. For the same reason, swapping $sq_2[i]$ results in operation $sq_2[i]'$ such that $sq_2[i]'.\text{pos} = sq_2[i].\text{pos} - i$.

Let $ipos = sq_2[i]'.\text{pos}$ and $jpos = sq_1[j]'.\text{pos}$. With every $sq_1[j]$, as in lines 5-8, we fix cursor j on sq_1 and move cursor i on sq_2 until we have identified all operations in sq_2 whose effect objects precede that of $sq_1[j]$. Then, as in line 10, we shift $sq_1[j].\text{pos}$ to account for those i operations. There are two cases in which $sq_2[i] \prec sq_1[j]$: 1) $ipos < jpos$, since now they are defined in the same state and can be compared; and 2) $ipos = jpos$ and $sq_2[i].\text{id} < sq_1[j].\text{id}$, meaning that there is a tie and we break the tie by comparing their site ids. It is worth noting, however, that the tie cannot always be broken this way. In Section 7, we will discuss the correctness conditions.

Algorithm 4. mergeISQ(sq_1, sq_2): sq

```
 1: sq ← [ ]; i ← 0; j ← 0
 2: while (i < |sq2|) and (j < |sq1|) do
 3:   if (sq2[i].pos − i ≤ sq1[j].pos) then
 4:     sq ← sq · sq2[i]
 5:     i ← i + 1
 6:   else
 7:     sq1[j].pos ← sq1[j].pos + i
 8:     sq ← sq · sq1[j]
 9:     j ← j + 1
10:   end if
11: end while
12: for (; i < |sq2|; i++) do
13:   sq ← sq · sq2[i]
14: end for
15: for (; j < |sq1|; j++) do
16:   sq1[j].pos ← sq1[j].pos + i
17:   sq ← sq · sq1[j]
18: end for
19: return sq
```

4.5 Merge Two Sequences

Function mergeSQ(sq_1, sq_2) merges sequences sq_1 and sq_2 into a new sequence sq. The two input sequences are ordered by the effects relation and $sq_1 \mapsto sq_2$. The output sequence sq is *effect-equivalent* with $sq_1 \cdot sq_2$. That is, if sq_1 is defined in state s_0 and the state becomes s_1 after execution of $sq_1 \cdot sq_2$, then it will also become s_1 after execution of sq in s_0. Here, we only specify function mergeISQ as in Algorithm 4, which merges two insertion sequences. The mergeDSQ function for merging two deletion sequences can be specified similarly.

The structure of Algorithm 4 is similar to that of the classic two-way merge algorithm, since the two input sequences are ordered. However, some transformation is required so as to make comparisons between any two given operations $sq_1[j]$ and $sq_2[i]$. This is achieved as follows: First, we swap $sq_1[j]$ to the end of sq_1. In this process, $sq_1[j].\text{pos}$ remains as-is. Second, we swap $sq_2[i]$ to the front of sq_2. In this process, $sq_2[i].\text{pos}$ is shifted to account for the i operations on its left, resulting in $sq_2[i]'.\text{pos} = sq_2[i].\text{pos} - i$. The reasons are similar to what have been explained above in Sections 4.3 and 4.4.

After the above transformation, $sq_1[j] \mapsto sq_2[i]'$, the two insertions are defined in two adjacent states, respectively, and we can compare their positions. Because they are insertions that are executed in tandem, if $sq_2[i]'.\text{pos} \leq sq_1[j].\text{pos}$, the effect object of $sq_2[i]'$ should precede that of $sq_1[j]$. Hence, in the resulting sequence, $sq_2[i]$ should be ordered before $sq_1[j]$. On the other hand, if $sq_2[i]'.\text{pos} > sq_1[j].\text{pos}$, the effect object of $sq_2[i]'$ should go after that of $sq_1[j]$, and hence, $sq_2[i]$ should be ordered after $sq_1[j]$ in the resulting sequence. However, to

account for the i operations in sq_2 that have been placed on its left, we need to adjust $sq_1[j].\text{pos}$ by i, as in line 7.

5 Obtaining Transactional Sequences

At each site, remote transactions are not merged until the current local transaction is ended and submitted. We first overview how to obtain transactional sequences. Then, we present related algorithms.

5.1 Overview

Some work could be done at the user interface level to help the user make informed decisions as to when the current local transaction should be ended and submitted and when remote transactions should be integrated. For example, previous work [27] proposes to visualize approximately which regions in a file have been modified by local and remote users. By heuristics, concurrent edits that happen on overlapping regions may conflict and should be merged sooner. Based on this information, the user can roughly tell whether the remote transactions are interesting or not and when to perform merging.

In general, there are two approaches to obtaining a transactional sequence T. First, in case that the system does not provide mechanisms for tracking exactly what update operations are performed by the user, diffing [28] can be used to compute an edit script. The resulting edit script on a text file is a sequence of insertions and deletions ordered by their positions. An algorithm can be devised to transform the script into a sequence $T = T_i \cdot T_d$ in which T_i and T_d are ordered by the effects relation. This approach is taken in our early work [27]. The disadvantages are that diffing can be expensive and an edit script is not the exact user operations.

Second, in case the system is able to track the exact update operations, we record them in buffer T that is maintained as $T_i \cdot T_d$. Every time a new local operation o is performed, property $(T_i \cdot T_d) \mapsto o$ must hold; If $o.\text{type} = \text{del}$, we swap o with operations in T_d right to left until it can be added in the right position in T_d; if $o.\text{type} = \text{ins}$, we first swap o with T_d, and then, with T_i right to left until it can be added in the right position of T_i. When a transaction is ended, function endTransaction(T, H) is invoked to merge T into the local history H, and meanwhile, transform T with H. After that, the resulting sequence T' is propagated to remote sites. We omit the algorithms for integrating a local operation into T due to their simplicity.

5.2 Submit a Local Transaction

Function *endTransaction* is specified in Algorithm 5. The two input sequences are $T = T_i \cdot T_d$ and $H = H_i \cdot H_d$, each of the four subsequences being ordered by the effects relation. Since no new remote transaction has been integrated before this function is called, the history H does not include operations concurrent with T. The precondition is $H \mapsto T$. When the function returns, T is merged into H', $T' = T'_i \cdot T'_d$ and $H' = H'_i \cdot H'_d$. The postconditions are that H' is effect-equivalent with $(H \cdot T)$ and the effects of H_d are excluded from T'. Note that the latter satisfies the precondition of Algorithm 1. Now T' is ready for propagation.

In Algorithm 5, we first swap H_d and T_i to obtain T'_i and H''_d. Since $H_i \mapsto H_d \mapsto T_i \mapsto T_d$, we have $H_i \mapsto T'_i \mapsto H''_d \mapsto T_d$. Then, we swap H''_d with T_d to obtain T'_d and H'''_d, by which we have $H_i \mapsto T'_i \mapsto T'_d \mapsto H'''_d$. Now we have achieved the first goal of the algorithm, i.e., to obtain $T' = T'_i \cdot T'_d$ in which the effects of all deletions that happen before T are excluded. Next, we merge T'_i into H_i, yielding H'_i, and merge T_d into H''_d, yielding H'_d. As discussed in Section 4.5, the preconditions of these two merging steps are $H_i \mapsto T'_i$ and $H''_d \mapsto T_d$, respectively, which are satisfied. As a result, we fulfill the second goal of Algorithm 5, i.e., to merge T into H.

Algorithm 5. endTransaction(T, H): (T', H')
1: $(T'_i, H''_d) \leftarrow \text{swapSQdi}(H_d, T_i)$
2: $(T'_d, H'''_d) \leftarrow \text{swapSQdd}(H''_d, T_d)$
3: $H'_i \leftarrow \text{mergeISQ}(H_i, T'_i)$
4: $H'_d \leftarrow \text{mergeDSQ}(H''_d, T_d)$
5: return $(T'_i \cdot T'_d,\ H'_i \cdot H'_d)$

5.3 Swap Two Sequences

Now, we specify the two swap functions called in Algorithm 5. Not to be tedious, we only explain function *swapSQdi* for swapping a deletion sequence and an insertion sequence. Function *swapSQdd* for swapping two deletion sequences can be defined similarly.

As in Algorithm 6, function swapSQdi(sq_2, sq_1) swaps the execution order of the two input sequences. The precondition is $sq_2 \mapsto sq_1$, where operations in each sequence are of the same type and ordered by the effects relation. The postcondition is $sq'_1 \mapsto sq'_2$.

Algorithm 6. swapSQdi$(sq_2, sq_1) : (sq'_1, sq'_2)$
1: $i \leftarrow 0$; $j \leftarrow 0$; $sq'_1 \leftarrow sq_1$; $sq'_2 \leftarrow sq_2$
2: **while** $(i < |sq_2|)$ and $(j < |sq_1|)$ **do**
3: **if** $(sq_2[i].pos < sq_1[j].pos - j)$ **then**
4: $sq'_2[i].pos \leftarrow sq'_2[i].pos + j$
5: $i \leftarrow i + 1$
6: **else**
7: $sq'_1[j].pos \leftarrow sq'_1[j].pos + i$
8: $j \leftarrow j + 1$
9: **end if**
10: **end while**
11: **for** (; $j < |sq_1|$; j++) **do**
12: $sq'_1[j].pos \leftarrow sq'_1[j].pos + i$
13: **end for**
14: **for** (; $i < |sq_2|$; i++) **do**
15: $sq'_2[i].pos \leftarrow sq'_2[i].pos + j$
16: **end for**
17: return (sq'_1, sq'_2)

Conceptually, we would have to swap every $sq_1[j]$ with the entire sq_2 right to left. However, this can be simplified because the two sequences are ordered by the effects relation. For any given $sq_1[j]$ and $sq_2[i]$, we do the following two steps: First, swap $sq_2[i]$ to the end of sq_2. Because its effect object precedes those of the operations on its right, its position remains as-is, i.e., $sq_2[i]'.\text{pos} = sq_2[i].\text{pos}$. Second, swap $sq_1[j]$ to the front of sq_1. Because its effect object follows those of the operations on its left, its position must be shifted by j to account for those j operations, i.e., $sq_1[j]'.\text{pos} = sq_1[j].\text{pos} - j$. As a result, $sq_2[i]' \mapsto sq_1[j]'$ and their positions can be compared.

Now we swap these two operations. Let $o_i = sq_2[i]'$ and $o_j = sq_1[j]'$. The conditions are $o_i.\text{type} = \text{del}$, $o_j.\text{type} = \text{ins}$, and $o_i \mapsto o_j$. We need to consider three cases:

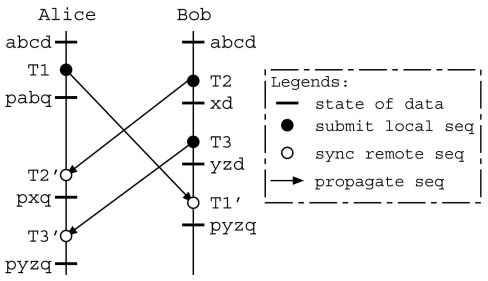


Fig. 2. Alice submitted sequence T_1 and Bob submitted two sequences T_2 and T_3.

First, if $o_i.\text{pos} < o_j.\text{pos}$, it means that after o_i.c is deleted, o_j.c is inserted on the right side of position $o_i.\text{pos}$. That is, $o_i \prec o_j$. Hence, if o_j is executed earlier than o_i, the position of o_j should be shifted because o_i.c is not deleted yet, whereas $o_i.\text{pos}$ should remain as-is because the earlier execution of o_j on the right does not change the position of o_i.c.

Conversely, if $o_i.\text{pos} > o_j.\text{pos}$, it means that $o_j \prec o_i$. Hence, if o_j is executed earlier, $o_j.\text{pos}$ should remain the same because the deferred execution of o_i has no impact, whereas $o_i.\text{pos}$ should be shifted because o_j.c has been inserted on the left of o_i.c.

On the other hand, if $o_i.\text{pos} = o_j.\text{pos}$, it means that, after o_i.c is deleted, o_j.c is inserted at the same position. By analysis in Section 5.2, we know that $o_i \rightarrow o_j$. According to proofs in [12], under this condition, we can safely mandate the effects relation such that the inserted object precedes the deleted object, that is, $o_j \prec o_i$. As a result, if we swap the execution order and o_j is executed earlier, $o_j.\text{pos}$ should remain as-is, whereas $o_i.\text{pos}$ is shifted.

Therefore, in Algorithm 6, the condition in line 3 corresponds to the first case, $o_i \prec o_j$, by which $o_j.\text{pos}$ should be shifted; the condition in line 6 corresponds to the second and third cases, $o_j \prec o_i$, by which $o_i.\text{pos}$ should be shifted.

We explain the loop in lines 2-10 as follows: On the one hand, in lines 6-8, for a given o_i in sq_2, scan sq_1 left to right until all operations that precede o_i in the effects relation are found; for each o_j: $o_j \prec o_i$, we do not need to shift $o_j.\text{pos}$ with regard to o_i; however, we need to account for the i operations in sq_2 that have already been in place in sq_2' and whose effects precede o_j; hence, we still need to shift $o_j.\text{pos}$ by i positions (as in line 7).

On the other hand, similarly, in lines 3-5, for a given o_j in sq_1, scan sq_2 left to right until all operations that precede o_j in the effects relation are found, and for each o_i such that $o_i \prec o_j$, shift $o_i.\text{pos}$ by j to account for those j operations that have already been in place (line 4).

6 An Example

As an example, consider a scenario in which two users, Alice and Bob, are coauthoring a document. Suppose that the initial state is a string "*abcd*." As shown in Fig. 2, Alice submitted one transaction T_1 and Bob submitted two transactions T_2 and T_3. The temporal relation between them is $T_1 \parallel (T_2 \rightarrow T_3)$. In the following, we study how these transactions are processed at local sites and integrated at remote sites.

6.1 Processing Local Transactional Sequence

Site A. Alice changes the document from "*abcd*" to "*pabq*" by performing the following sequence of operations: $[ins(0,p), del(4), del(3), ins(3,q)]$. As the operations are performed, the sequence is reordered into an effect-equivalent transactional sequence T_1:

$$T_1 = [ins(0,p), ins(3,q), del(4), del(4)].$$

When T_1 is submitted, the local history is empty. As a result, $H^A = T_1$ and T_1 is propagated as-is.

Site B. Bob first changes "*abcd*" into "*xd*" by performing $[del(1), del(1), ins(1,x), del(0)]$. It is reordered into a transactional sequence T_2 (operations in which accidentally have the same position parameters):

$$T_2 = [ins(1,x), del(0), del(1), del(1)].$$

When T_2 is submitted, the local history is empty. Hence, we get $H^B = T_2$ and propagate T_2 as-is.

Later, Bob changes "*xd*" into "*yzd*" by performing the following sequence: $[ins(1,z), del(0), ins(0,y)]$. It is reordered into a transactional sequence T_3:

$$T_3 = [ins(0,y), ins(2,z), del(1)].$$

Then, Algorithm 5 is called to integrate T_3 into local history H^B and produce a transformed transactional sequence that is to be propagated to site A. As a result, the effects of deletions in the local history are excluded from T_3 and T_3 is merged into the history. The resulting T_3 and history $H^B = H_i^B \cdot H_d^B$ are as follows:

$$\begin{aligned} T_3 &= [ins(0,y), ins(3,z), del(2)], \\ H_i^B &= [ins(0,y), ins(2,x), ins(3,z)], \\ H_d^B &= [del(1), del(1), del(2), del(2)]. \end{aligned}$$

6.2 Processing Remote Transactional Sequence

Site A. Algorithm 1 is called to integrate T_2, which transforms it into an executable form T_2' and merges it with H^A yielding $H^{A'} = H_i^{A'} \cdot H_d^{A'}$, as follows:

$$\begin{aligned} T_2' &= [ins(2,x), del(1), del(2)], \\ H_i^{A'} &= [ins(0,p), ins(2,x), ins(4,q)], \\ H_d^{A'} &= [del(1), del(2), del(3), del(3)]. \end{aligned}$$

After executing T_2' on Alice's document, it becomes "*pxq*." Later, when T_3 is integrated, the results T_3' and $H^{A''} = H_i^{A''} \cdot H_d^{A''}$ are as follows:

$$\begin{aligned} T_3' &= [ins(1,y), ins(3,z), del(2)], \\ H_i^{A''} &= [ins(0,p), ins(1,y), ins(3,x), ins(4,z), ins(6,q)], \\ H_d^{A''} &= [del(2), del(2), del(3), del(4), del(4)]. \end{aligned}$$

Executing T_3' on Alice's document yields "*pyzq*."

Site B. When T_1 is integrated, the results T_1' and history $H^{B'} = H_i^{B'} \cdot H_d^{B'}$ are as follows:

$$\begin{aligned} T_1' &= [ins(0,p), ins(3,q), del(4)], \\ H_i^{B'} &= [ins(0,p), ins(1,y), ins(3,x), ins(4,z), ins(6,q)], \\ H_d^{B'} &= [del(2), del(2), del(3), del(4), del(4)]. \end{aligned}$$

Executing T_1' changes Bob's document to "*pyzq*," which is identical with the final state of Alice.

7 Analysis of Correctness

In this section, we explain why the presented algorithm is correct and why it is so designed. Our early work has incrementally built formal, provable correctness conditions for OT algorithms [16], [21], [12]. In particular, our latest theoretical work [12], [13] establishes the following two formal conditions, which implies convergence:

1. Causality Preservation: the happens-before relation between operations is always maintained whenever an operation is executed.
2. Admissibility Preservation: the execution of every operation is *admissible*, i.e., it does not introduce inconsistent ordering of objects in the shared data at different sites.

Condition (1) is satisfied by using vector time stamps. This part is omitted from the algorithm presentation since it is well understood [7], [10]. To satisfy condition (2), our approach is to first prove sufficient conditions of the basic IT and swap functions, and then, design a control procedure that ensures them while integrating local and remote operations. Assuming that the two input operations o_1 and o_2 are admissible, we have proved the following two sufficient conditions [12], [13]:

1. IT(o_1, o_2) is admissible if $o_1 \sqcup o_2$ and in case they are both insertions and $o_1.pos = o_2.pos$, $o_1 \parallel o_2$ and neither includes effects of any deletions.
2. swap(o_1, o_2) is admissible if $o_1 \mapsto o_2$ and in case o_1 is a deletion and o_2 is an insertion, and $o_1.pos = o_2.pos$, $o_1 \rightarrow o_2$ holds and o_2 has not been transformed with any concurrent operations.

When presenting the algorithms, we have explained what the preconditions are and how they are satisfied in the main steps. More formally, we prove the correctness of ABST through the following lemmas and theorem:

Lemma 7.1. *The invocations of the sequence transformation function ITSQ are admissible in ABST.*

Proof. The purpose of ITSQ(sq_1, sq_2), where $sq_1 \sqcup sq_2$, is to incorporate the effects of sq_2 in sq_1 such that $sq_2 \mapsto sq_1'$ holds for the resulting sequence sq_1'. Every operation $sq_1[j]$ is transformed by a series of calls to the above two basic swap and IT functions, as follows:

1. swap $sq_1[j]$ to the front of sq_1;
2. inclusively transform $sq_1[j]$ with operations in sq_2;
3. swap $sq_1[j]$ back to its original position in sq_1.

Take function ITSQii (Algorithm 3) as the example. Step 1 is achieved by line 4; step 2 is achieved in lines 5-12 and 14-16 which shift the position of $sq_1[j]$ only on those operations in sq_2 whose effects precede that of $sq_1[j]$; and step 3 is implied in line 10.

Since swap is always called between adjacent operations in sq_1, which are of the same type, steps 1 and 3 are admissible by condition 2.

In Algorithm 1 when ITSQ(T_i, csq_i) is called (line 2), $T_i \sqcup csq_i$, $T_i \parallel csq_i$, and insertions in neither T_i nor csq_i include effects of any deletions. Hence, by condition 1, it is safe to break the position tie between two insertions and the IT in step 2 is admissible. When the other ITSQs are called in Algorithm 1 (lines 3, 5, 7, and 8), at least one of the input sequences is deletion. Therefore, their ITs in step 2 are admissible also by condition 1. □

Lemma 7.2. *The invocations of the sequence swapping function swapSQ are admissible in ABST.*

Proof. The proof is similar to that of Lemma 7.1 as above. Take function swapSQdi (Algorithm 6) as the example. Function swapSQdi(sq_2, sq_1), where $sq_2 \mapsto sq_1$, is essentially a series of calls to the basic swap and IT functions: For every $sq_1[j]$, 1) swap it with operations in sq_2 and 2) incorporate the effects of operations in sq_2 that precede the effect of $sq_1[j]$. The latter is actually not the exact basic IT function but a variation of IT.

Since the two input sequences are already ordered, part 1) can be simplified by swapping $sq_1[j]$ to the front of sq_1 and swapping $sq_2[i]$ to the end of sq_2. When swapSQdi(H_d, T_i) is called in Algorithm 5, T_i is a newly generated transactional sequence, the operations in T_i have not been transformed with any concurrent operations. Hence, $H \mapsto T$ and $H \rightarrow T$ hold. These swap operations are admissible by condition 2.

Part 2) is semantically the same as inclusion transformation (IT) since it incorporates the effects of sq_2 into $sq_1[j]$. However, the difference is that the two operations $sq_1[j]$ and $sq_2[i]$ involved in every step satisfy $sq_2[i] \mapsto sq_1[j]$ rather than $sq_2[i] \parallel sq_1[j]$. The upside is that, because $sq_2[i] \mapsto sq_1[j]$, there is no ambiguity when determining the effects relation of these two operations. The case of $sq_2[i] \parallel sq_1[j]$ is actually more complicated in IT, as suggested in condition 1, because the effects relation is ambiguous when the two insertions tie. As a result, the output of part 2) is also admissible. □

Lemma 7.3. *The invocations of the sequence merging function mergeSQ are admissible in ABST.*

Proof. The proof is similar to those of Lemmas 7.1 and 7.2 since the merging process is essentially also a series of calls to the basic swap function and a variation of the basic IT function. Hence, we omit the details. □

Theorem 7.4. *The* ABST *algorithm is correct with regard to 1) the causality preservation condition and 2) the admissibility preservation condition.*

Proof. The ABST algorithm mainly consists of two parts: processing a local transactional sequence (Algorithm 5) and processing a remote transactional sequence (Algorithm 1). As explained in Section 4, when a remote transactional sequence T is integrated, it must be causally ready, which satisfies condition 1). By Lemmas 7.1, 7.2, and 7.3, invocations of all the key steps ITSQ, swapSQ, and mergeSQ in Algorithms 5 and 1 are admissible. Hence, condition 2) is also satisfied. □

The sufficient condition 1 for IT explains why we maintain the history H as $H_i \cdot H_d$ at every site. One of the most challenging problems in OT is how to break ties in IT functions, as revealed in [10], [16], [12], [21], [20]. By

maintaining H this way, we ensure the correctness of IT [12]. As analyzed in Sections 8 and 2, this does not undermine efficiency. In fact, ABST is more efficient than its competitors even though H is so maintained.

8 Complexities and Performance

The space complexity of ABST is trivially $O(|H| + |T|)$. The time complexity of integrating either a local transaction or a remote transaction is $O(|H| + |T|)$.

As argued in Section 2.3.2, we chose to compare the performance of ABST against our own prior work ABT [12], [13] because ABT is formally proved, fully presented, and more efficient than its competitors. Nevertheless, its time complexity is $O(|H|^3)$ when integrating a remote sequence T with history H, and $|T|$ and $|H|$ are comparable. To obtain a more concrete sense of what the performance implications are in real applications, we conducted the following experiments.

We implemented ABT and ABST in C#. The experiments are performed on two mobile devices:

- *Experiments on N810:* The algorithms are compiled using Maemo Mono C# and executed on a Nokia N810 tablet, with a 400 MHz ARM v61 CPU (TI OMAP 2,420) and 128 MB DDR RAM. The OS is Maemo Linux with kernel 2.6.21-omap1.
- *Experiments on nx5000:* The algorithms are compiled using Linux Mono C# and executed on an HP laptop Compaq nx5000, with a 1.50 GHz Intel(R) Pentium(R) M CPU and 512 MB DDR RAM. The OS is Arch Linux with kernel 2.6.28.

The experiments are designed to study how long it takes to integrate a remote sequence T, which is dominated by function Integrate(T, H). In each experiment, we generate two sequences H and T such that $T \parallel H$. Then, ABT and ABST are called to integrate T with H, respectively. The positions of the operations in each sequence are roughly uniformly distributed over the shared document D. That is, each operation is generated with probability $\frac{1}{|D|}$ in any position in the current document and $|D| \gg |H| + |T|$. The size of each sequence is varied from 0 to 3,000, with step 300. The percentage of insertions in each sequence is 20, 50, or 80 percent, implying that the deletion ratio is accordingly 80, 50, or 20 percent. For every given T and H, Integrate(T, H) is executed five times and the average time is recorded.

The experimental results for the N810 tablet are shown in Fig. 3. With similar observations, we omit the curves for the nx5000 laptop. Some selected data points for both devices are shown in Table 2. The data clearly show that ABST is faster than ABT by several orders of magnitude. For example, when $|T| = 3{,}000$, $|H| = 3{,}000$, and the insertion ratio is 80 percent, it takes 3,563,556 ms or over 59 minutes in ABT versus 1,452 ms or less than 1.5 seconds in ABST to integrate T with H on N810.

Our data suggest that ABST is efficient enough to support a range of mobile and asynchronous collaboration tasks. Intuitively, an algorithm that is efficient to integrate a long sequence must be efficient to integrate a short sequence or a single operation. The time complexity for the latter case is $O(|H|)$ as well, which excels most existing OT algorithms that take $O(|H|^2)$ or more time to integrate one remote operation. That is, ABST is also efficient enough for real-time collaboration tasks that feature frequent propagation and integration of operations in small batches [23]. Therefore, ABST can facilitate a wide spectrum of synchronous to asynchronous tasks.

9 Discussion: Conflict Detection

Consistency maintenance in collaborative applications occurs at two levels, namely, syntactic and semantic consistency [8]. In general, the former addresses how to achieve the same view of shared data at all sites, whereas the latter further constrains the converged view such that it makes sense in the context of specific applications. Traditionally, the scope of OT has been limited to syntactic consistency [8] and so is this work. Detection and resolution of conflicts are part of semantic consistency control. Policies and mechanisms for conflict handling are necessarily application-specific. Different types of files, such as research articles, software programs, and Web pages, have different requirements for semantic consistency, e.g., grammar rules.

In real life, there is usually a division of labor and the users follow well-understood procedures or social conventions when making changes to shared files. For example, when coauthoring an article, a user would consciously avoid modifying a section if she knows that a collaborator is working on that section. Hence, concurrent updates to exactly the same word or sentence do not happen frequently in practices [30].

Nevertheless, we follow the same direction as in previous OT works by augmenting algorithms with user interfaces. For example, as in [27], when a remote sequence is received (and before it is merged), the regions of the file modified by remote and local sequences are marked in different colors. Based on the visual cues, the user can often easily tell approximately whether there are conflicts or not and when to resolve them.

Furthermore, we can provide additional interfaces for users to define their own conflict detection policies. Detection of conflicts can happen either before merging, as explained above, or after (fully automated) merging. Our method assumes that the probability of semantic conflicts increases as the positions of two concurrent updates get closer to each other. Since this is not the focus of this paper, we only outline the ideas, as follows:

First, at the macroscopic level, we consider two concurrent sequences sq_1 and sq_2 for the same file, each being of the same operation type and $sq_1 \sqcup sq_2$. The operations are ordered by the effects relation. Let $a_1 = sq_1[0].\text{pos}$, $b_1 = sq_1[|sq_1| - 1].\text{pos}$, $a_2 = sq_2[0].\text{pos}$, and $b_2 = sq_2[|sq_2| - 1].\text{pos}$. Then, the spatial spans of sq_1 and sq_2 are $L_1 = b_1 - a_1 + 1$ and $L_2 = b_2 - a_2 + 1$, respectively. The distance between their boundaries is $d = |\frac{1}{2}(a_2 + b_2) - \frac{1}{2}(a_1 + b_1)| - \frac{1}{2}L_1 - \frac{1}{2}L_2$. In particular, if $d \leq 0$, then sq_1 and sq_2 touch or overlap each other. Intuitively, the smaller the distance d, the higher the probability of the potential conflicts. The user can define an application-specific threshold $\mathcal{D}(\mathcal{D} \geq 0)$. If $d \leq \mathcal{D}$, the users are notified of potential conflicts on the file and the two conflicting regions are highlighted.

Second, at the microscopic level, from two given sequences sq_1 and sq_2 that satisfy the same conditions as above, we can pick up every pair of operations $sq_1[i]$ and $sq_2[j]$ that potentially conflict. For example, suppose that

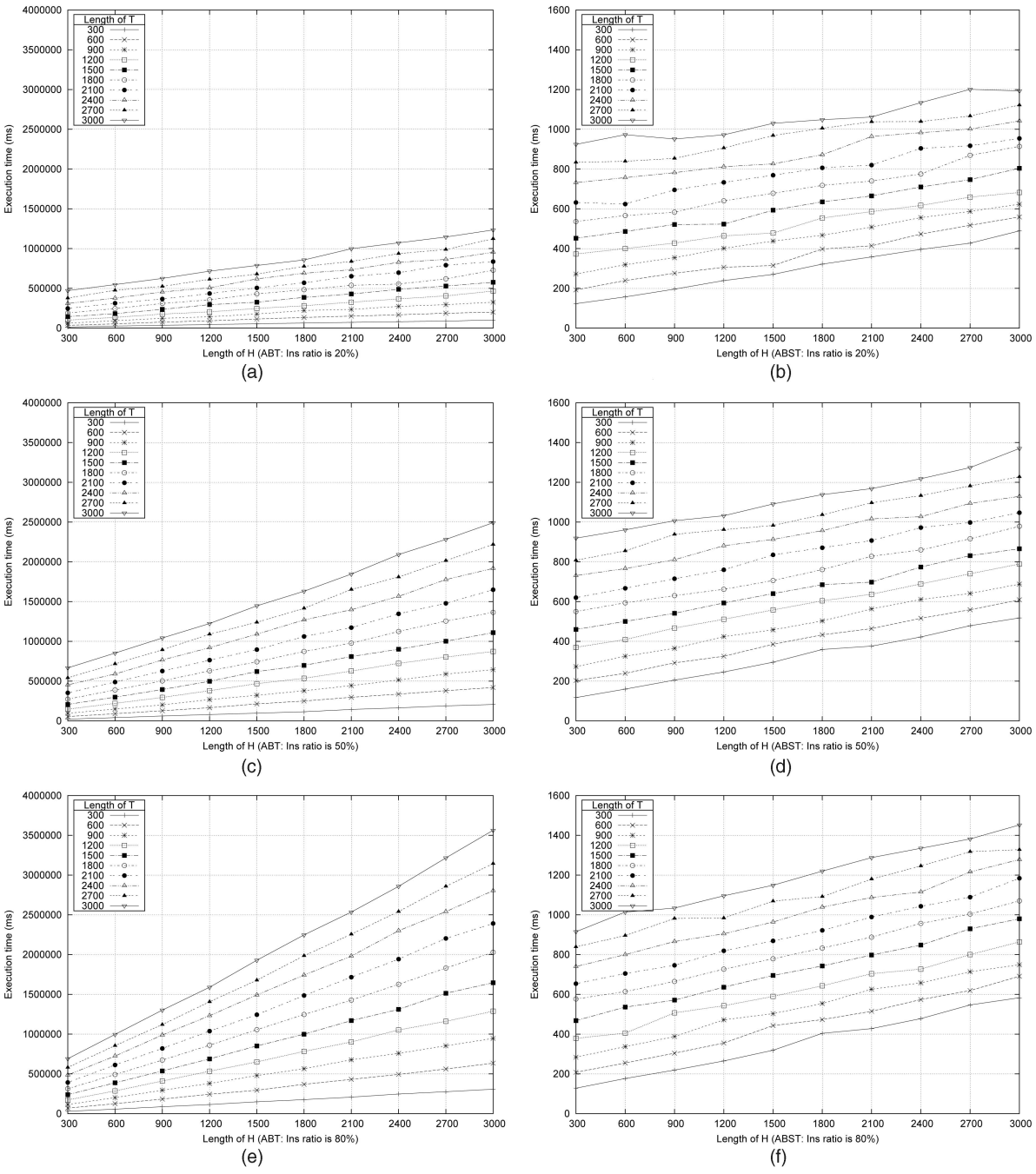


Fig. 3. The time to integrate T with H on a Nokia N810 tablet using ABT and ABST under varying insertion rations: (a) ABT with 20 percent insertions, (b) ABST with 20 percent insertions, (c) ABT with 50 percent insertions, (d) ABST with 50 percent insertions, (e) ABT with 80 percent insertions, and (f) ABST with 80 percent insertions.

they are both insertions. Similarly to Algorithm 3, we first swap them to the front of sq_1 and sq_2, respectively. Let $ipos = sq_1[i].\text{pos} - i$ and $jpos = sq_2[j].\text{pos} - j$. Consider their distance $d = |ipos - jpos|$. When $d \leq \mathcal{T}$, where $\mathcal{T}$ is a user-specified threshold, there is a potential conflict between the two operations. All those pairs can be collected and reported to the user graphically.

Note that the above thresholds $\mathcal{D}$ and $\mathcal{T}$ are integers that account for the number of characters. With some extensions, the algorithms and thresholds can also address strings and semantic units such as words, sentences, paragraphs, and sections. The user can define rules for conflict detection in a continuous spectrum of granularity in terms of distance between update positions.

10 Conclusions

This paper presents an efficient OT-based algorithm called ABST for merging sequences of updates to the same file. ABST improves the time complexity of the state of the art from $O(|H|^3)$ to $O(|H|)$, where H is the operation history. Practically, it can take hours in previous work versus seconds

TABLE 2
A Summary of the Experimental Results

$\|H\|$	$\|T\|$	Ins(%)	Time(ms)			
			N810		nx5000	
			ABT	ABST	ABT	ABST
300	300	20	13833	123	558	5
3000	300	20	102403	489	4444	18
300	3000	20	471033	924	21054	47
3000	3000	20	1233955	1193	56924	53
300	300	50	22573	117	909	5
3000	300	50	205777	518	8571	20
300	3000	50	666074	919	29096	46
3000	3000	50	2491809	1370	112778	64
300	300	80	29627	128	1244	5
3000	300	80	307202	583	13222	23
300	3000	80	691217	915	30009	48
3000	3000	80	3563556	1452	159668	75

in this work to merge two long sequences on a mobile device. The linear complexity of ABST makes it possible to support a wide spectrum of synchronous to asynchronous collaboration tasks on resource-constrained platforms such as cell phones. The algorithm can be used in custom collaborative applications that are able to track editing operations (e.g., [10]) as well as legacy applications that are adapted to be collaborative without modifying source code (e.g., [19], [27]). Based on the algorithm, it is possible to define a spectrum of policies for conflict detection.

The focus of this paper is on the efficiency aspect of the presented ABST algorithm, not how it can be effectively used in practical applications such as collaborative editing systems. We plan to extend this work along two directions: First, on the theoretical side, we will extend the algorithm to support selective undo of any operation or sequence, which is a useful building block for error recovery and conflict resolution [20], [31]. Second, on the application side, it is possible to build the extended algorithms into industry products on mobile and Web platforms. In the background of specific applications, we will be in a better position to research on domain-specific conflict handling mechanisms and study their usability.

ACKNOWLEDGMENTS

The authors thank the anonymous expert reviewers for their very insightful and constructive comments. Yiming Ma and Guang Yang at Nokia Research Center, Palo Alto, and Chengzheng Sun at Nanyang Technological University, Singapore, also provided valuable feedback. The work is supported 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, the Shanghai Science & Technology Committee Key Fundamental Research Project under Grant No. 08JC1402700, and the Shanghai Leading Academic Discipline Project under Grant No. B114. This paper is substantially extended from its conference version [1].

REFERENCES

[1] B. Shao, D. Li, and N. Gu, "A Sequence Transformation Algorithm for Supporting Cooperative Work on Mobile Devices," *Proc. ACM Conf. Computer-Supported Cooperative Work (CSCW),* pp. 159-168, Feb. 2010.

[2] Y. Saito and M. Shapiro, "Optimistic Replication," *ACM Computing Survey,* vol. 37, no. 1, pp. 42-81, Mar. 2005.

[3] J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," *ACM Trans. Computer Systems,* vol. 10, no. 1, pp. 3-25, Feb. 1992.

[4] K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and A.J. Demers, "Flexible Update Propagation for Weakly Consistent Replication," *Proc. 16th ACM Symp. Operating Systems Principles,* pp. 288-301, 1997.

[5] R. Guy, P. Reiher, D. Ratner, M. Gunter, W. Ma, and G. Popek, "Rumor: Mobile Data Access through Optimistic Peer-to-Peer Replication," *Proc. 17th Int'l Conf. Conceptual Modeling (ER '98): Workshop Mobile Data Access,* 1998.

[6] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel, "The IceCube Approach to the Reconciliation of Divergent Replicas," *Proc. 20th Ann. ACM Symp. Principles of Distributed Computing (PODC '01),* pp. 210-218, 2001.

[7] C.A. Ellis and S.J. Gibbs, "Concurrency Control in Groupware Systems," *Proc. ACM SIGMOD '89,* pp. 399-407, 1989.

[8] C. Sun and C. Ellis, "Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements," *Proc. ACM Conf. Computer-Supported Cooperative Work,* pp. 59-68, Dec. 1998.

[9] C.M. Hymes and G.M. Olson, "Unblocking Brainstorming through the Use of Simple Group Editor," *Proc. ACM Conf. Computer-Supported Cooperative Work (CSCW '92),* pp. 99-106, Nov. 1992.

[10] 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 Trans. Computer-Human Interaction,* vol. 5, no. 1, pp. 63-108, Mar. 1998.

[11] A.H. Davis, C. Sun, and J. Lu, "Generalizing Operational Transformation to the Standard General Markup Language," *Proc. ACM Conf. Computer-Supported Cooperative Work (CSCW '02),* pp. 58-67, Nov. 2002.

[12] R. Li and D. Li, "Commutativity-Based Concurrency Control in Groupware," *Proc. First IEEE Conf. Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom '05),* Dec. 2005.

[13] D. Li and R. Li, "An Admissibility-Based Operational Transformation Framework for Collaborative Editing Systems," to be published in *Computer Supported Cooperative Work: The J. Collaborative Computing,* vol. 19, no. 1, pp. 1-43, 2010.

[14] S. Balasubramaniam and B.C. Pierce, "What Is a File Synchronizer?" *Proc. ACM/IEEE MobiCom '98,* pp. 98-108, 1998.

[15] V. Gaburici, P. Keleher, and B. Bhattacharjee, "File System Support for Collaboration in the Wide Area," *Proc. 26th IEEE Int'l Conf. Distributed Computing Systems (ICDCS '06),* p. 26, 2006.

[16] D. Li and R. Li, "Preserving Operation Effects Relation in Group Editors," *Proc. ACM Conf. Computer-Supported Cooperative Work (CSCW '04),* pp. 457-466, Nov. 2004.

[17] Y.-W. Lee, K.-S. Leung, and M. Satyanarayanan, "Operation-Based Update Propagation in a Mobile File System," *Proc. USENIX Ann. Technical Conf.,* June 1999.

[18] M. Suleiman, M. Cart, and J. Ferrié, "Concurrent Operations in a Distributed and Mobile Collaborative Environment," *Proc. IEEE Int'l Conf. Data Eng. (ICDE '98),* pp. 36-45, Feb. 1998.

[19] 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 Trans. Computer-Human Interaction,* vol. 13, no. 4, pp. 531-582, Dec. 2006.

[20] D. Sun and C. Sun, "Context-Based Operational Transformation in Distributed Collaborative Editing Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 20, no. 10, pp. 1454-1470, Oct. 2009.

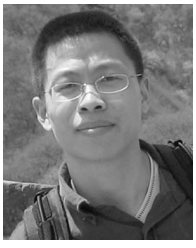
[21] R. Li and D. Li, "A New Operational Transformation Framework for Real-Time Group Editors," *IEEE Trans. Parallel and Distributed Systems,* vol. 18, no. 3, pp. 307-319, Mar. 2007.

[22] G. Oster, P. Urso, P. Molli, and A. Imine, "Proving Correctness of Transformation Functions in Collaborative Editing Systems," Technical Report 5795, INRIA, Dec. 2005.

[23] 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, vol. 7, no. 4, pp. 55-61, Oct.-Dec. 2000.

[24] D. Li and R. Li, "A Performance Study of Group Editing Algorithms," *Proc. 12th Int'l Conf. Parallel and Distributed Systems (ICPADS '06),* pp. 300-307, July 2006.

[25] H. Shen and C. Sun, "Flexible Merging for Asynchronous Collaborative Systems," *Proc. Int'l Conf. Cooperative Information Systems (CoopIS '02)*, pp. 304-321, Oct. 2002.
[26] H. Shen and C. Sun, "A Log Compression Algorithm for Operation-Based Version Control Systems," *Proc. IEEE Int'l Computer Software and Application Conf.*, pp. 867-872, Aug. 2002.
[27] D. Li and J. Lu, "A Lightweight Approach to Sharing Heterogeneous Single-User Editors," *Proc. ACM Conf. Computer-Supported Cooperative Work (CSCW '06)*, pp. 139-148, Nov. 2006.
[28] E.W. Myers, "An $O(ND)$ Difference Algorithm and Its Variations," *Algorithmica*, vol. 1, pp. 251-266, 1986.
[29] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 7, pp. 558-565, July 1978.
[30] S. Noel and J.-M. Robert, "Empirical Study on Collaborative Writing: What Do Co-Authors Do, Use, and Like," *Computer Supported Cooperative Work: The J. Collaborative Computing*, vol. 13, pp. 63-89, 2004.
[31] B. Shao, D. Li, and N. Gu, "An Algorithm for Selective Undo of Any Operation in Collaborative Applications," *Proc. ACM Conf. Supporting Group Work (GROUP)*, to appear, Nov. 2010.



Bin Shao is currently an Associate Researcher at Microsoft Research Asia. He received the PhD degree from the School of Computer Science, Fudan University, China in June 2010. His research interests include computer-supported cooperative work, optimistic replication, operational transformation, and distributed systems. He is a member of the IEEE.

Du Li received the PhD degree in computer science from UCLA in 2000. He is a senior research scientist in Nokia Research Center, Palo Alto, California. He was a recipient of a CAREER Award from the National Science Foundation in 2002. His research interests include mobile systems, collaborative systems, distributed computing, Web 2.0, and computer-supported cooperative work. More information about his research is available at http://www.linkedin.com/in/lidu008. He is a member of the IEEE.

Ning Gu received the PhD degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, China, 1995. He is a professor and the director of the Cooperative Information and Systems Lab at the School of Computer Science, Fudan University, China. His current research interests include computer-supported cooperative work, data management, distributed systems, and social networking. More information about his research is available at http://cscw.fudan.edu.cn/. He is a member of the IEEE.