Optional and Responsive Fine-Grain Locking in Internet-Based Collaborative Systems

Chengzheng Sun

Abstract—Locking is a standard technique in distributed computing and database systems used to ensure data integrity by prohibiting concurrent conflicting updates on shared data objects. Internet-based collaborative systems are a special class of distributed applications which support human-to-human interaction and collaboration over the Internet. In this paper, a novel optional and responsive fine-grain locking scheme is proposed for consistency maintenance in Internet-based collaborative editors. In the proposed scheme, locking is made optional in the sense that a user may update any part of the document without necessarily requesting a lock, thus saving the users the burden of having to use locks while editing and the system the overhead of executing locking operations most of the time in a collaborative editing session. In the face of high communication latency in the Internet environment, responsive locking is achieved by granting the permit to the user for updating the data region immediately after issuing a locking request. Moreover, multiple fine-grain locks can be placed on different regions inside a document to allow concurrent and mutually exclusive editing on the same document. Protocols and algorithms for locking conflict resolution and consistency maintenance are devised to address special technical issues involved in optional and responsive fine-grain locking. The proposed locking scheme and supporting techniques have been implemented in an Internet-based collaborative editor to demonstrate its feasibility and usability.

Index Terms—Consistency maintenance, optional locking, responsiveness, operational transformation, collaborative editors, distributed systems, Internet computing.

1 Introduction

Internet-based collaborative systems are a special class of distributed applications which support human-to-human interaction and collaboration over the Internet [7], [9], [16], [21], [26]. We are particularly interested in Internet-based real-time collaborative editing systems which allow a group of users to view and edit the same text/graphics/image/multimedia document at the same time over the Internet [5], [6], [12], [14], [19], [21]. The goal of our research is to design and implement real-time collaborative editing systems meeting the following requirements [21]:

1. *high responsiveness*—the response to local user actions must be quick, ideally as quick as a single-user editor;
2. *high concurrency*—multiple users are allowed to concurrently edit any part of the document at any time; and
3. *ability to hide communication latency*—the system should work well in an environment with high and nondeterministic communication latency, such as the Internet.

These requirements have led us to adopt a replicated system architecture for the storage of shared documents: The shared documents are replicated at the local storage of each collaborating site. A collaborating site can be a PC or a workstation, consisting of a local user interface for generating local operations and for displaying local document state, a local storage for storing document replicas, and computing and communication facilities for processing and propagating operations. With the replicated architecture, it becomes possible for multiple users to concurrently edit their local copies of the shared document and to get their operations reflected on their local interfaces immediately. One of the most significant challenges in the design and implementation of replicated collaborative editing systems is consistency maintenance of replicated documents in the face of concurrent updates [21].

Locking is a standard technique in distributed computing and database systems to ensure data integrity by prohibiting concurrent conflicting updates on shared data objects [1]. Locking has also been used for consistency maintenance of shared documents in various collaborative editing systems [2], [10], [12], [13], [14], [15]. All existing locking schemes have one thing in common: Locking is *compulsory* in the sense that a lock *must* be requested before updating a data region.

In contrast to existing locking schemes, the locking scheme proposed in this paper is *optional* in the sense that a user may update any (unlocked) region without necessarily requesting a lock on it. If a lock has been placed on a region, however, a user can update this region only if she/he owns a lock covering the region. Locking is made optional because our research in collaborative editing has found that

1. there is no role for locking to play in maintaining *syntactic* consistency (see detailed discussions in Section 3, and
2. there is no need to use locks when multiple users are editing different regions most of the time in a collaborative editing session.

• *The author is with the School of Computing and Information Technology, Griffith University, Brisbane, Qld 4111, Australia.
E-mail: c.sun@cit.gu.edu.au.*

*Manuscript received 21 Dec. 2000; accepted 8 Jan. 2002.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 113341.*

Only when a user wishes to ensure her/his exclusive updating right to a region may she/he occasionally request a lock on a region. Under the optional locking scheme, it is possible that one user accidentally intrudes on an unlocked region being updated by another user. If this happens, the system can notify both users involved and let the users take proper actions to resolve the conflict. For example, one of the users could then place a lock on the region and repair the inconsistency (if any) by using system-provided *undo* facilities [24], [17], [19]. This way of handling conflict is reasonable since conflicts are rare in collaborative environments [4],and, if conflicts do occur, human users are capable of adjusting their actions and repairing temporary inconsistencies. The major advantage of optional locking is that it saves the users the burden of having to use locks while editing and the system the overhead of executing locking operations in most of the time in a collaborative editing session.

Apart from being optional, the proposed locking scheme is also highly *responsive* in the sense that a user requesting a lock on a region is immediately granted permission to edit the region without being blocked. High responsiveness is particularly important in the Internet environment with a high and nondeterministic communication latency. A major technical challenge in achieving highly responsive locking is how to resolve the conflict when multiple locks are requested concurrently on the same data region in the document.

Moreover, to support concurrent editing on the same document by multiple users, our locking scheme is *fine-grained* at the level down to individual characters in a document. In our locking scheme, users are allowed to place locks on any region of an arbitrary size inside a document, which is in contrast to coarse-grain locking on an entire file or document in traditional file systems and many collaborative systems. A special technical challenge in supporting fine-grain locking is how to maintain locking status consistency across all collaborating sites in the face of concurrent locking and editing operations.

In this paper, we will discuss the special technical issues and solutions involved in supporting optional and responsive fine-grain locking in Internet-based real-time collaborative text editing systems. The basic ideas of our work on optional locking have been presented in an earlier conference publication [23]. This paper is a significant revision and extension of the earlier version. Additional contributions in this paper include a new responsive locking scheme combined with protocols for resolving concurrent locking conflict (Section 4) and new algorithms for locking permission check and for realizing the new conflict resolution protocols (Section 6).

The rest of this paper is organized as follows: First, some basic concepts and definitions in collaborative text editing are introduced in Section 2. Next, the role of locking in maintaining two different classes of consistency in collaborative editing systems is examined in Section 3. A responsive locking scheme combined with protocols for resolving locking conflict is proposed and discussed in Section 4. Special technical issues and solutions involved in achieving consistent fine-grain locking are discussed in Section 5. Algorithms for realizing the responsive fine-grain locking scheme are presented in Section 6. The proposed locking scheme is compared to the traditional and alternative locking schemes in Section 7. Finally, major contributions and future work of our research are presented in Section 8.

2 Basics in Collaborative Text Editing

2.1 A Text Document Data Model

A text document (with no formatting) is modeled by a sequence of characters, referred to from 0 to the end of the document. Each primitive editing/locking operation on the document has one *position* parameter which specifies the *absolute* position in the document at which the operation is to be performed.

It should be pointed out that the above text document data model is just a conceptual view of the text document and it does not dictate the actual data structure which is used to implement the document state. This conceptual data model could be implemented in various different internal data structures, such as a single array of characters, the linked-list structures, the buffer-gap structure, and virtual-memory blocks [25].

2.2 Primitive Operations

The document state can only be changed by executing the following two primitive editing operations:

1. $Insert[S, P]$, which inserts string S at position P.
2. $Delete[N, P]$, which deletes N characters starting from position P.

It has been shown that practical text editing systems, such as *vi* and *Emacs*, can be implemented on top of these two primitives [10], [19], [21], [25]. It is assumed that each collaborating site maintains a *History Buffer (HB)* to keep track of all executed operations at that site. The current document state can be uniquely determined by applying the sequence of operations in the *HB* on the initial document state.

In addition to editing operations, the following two fine-grain locking operations may also be generated to lock or unlock any region of an arbitrary length in the document:

1. $Lock[N, P]$, which locks a region with N characters starting from position P.
2. $Unlock[N, P]$, which unlocks a region with N characters starting from position P.

It is assumed that each collaborating site maintains a *Locking Table (LT)* to keep track of executed *Lock* operations at that site. All entries in the *LT* together represent the current locking status of the document at that site.

2.3 Operation Relationship

Both editing and locking operations can be generated and executed in arbitrary orders in a collaborative editing environment. Following Lamport [11], the causal ordering relationship among all operations is defined in terms of their generation and execution sequences as follows:

Definition 1 (Causal ordering relation "$\rightarrow$"). *Given two operations O_a and O_b, generated at sites i and j, then $O_a \rightarrow$*

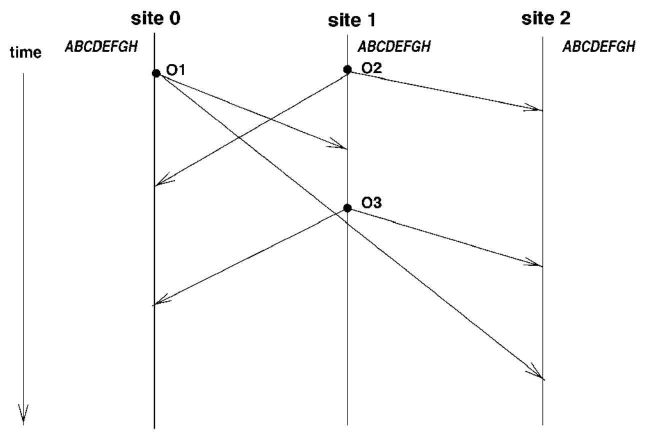

Fig. 1. A scenario of a real-time collaborative editing session. The initial document contains a string of characters "*ABCDEFGH*" and is replicated at all sites.

O_b iff: 1) $i = j$ and the generation of O_a happened before the generation of O_b or 2) $i \neq j$ and the execution of O_a at site j happened before the generation of O_b or 3) there exists an operation O_x, such that $O_a \rightarrow O_x$ and $O_x \rightarrow O_b$.

Definition 2 (Dependent and independent operations). *Given any two operations O_a and O_b. 1) O_b is dependent on O_a iff $O_a \rightarrow O_b$. 2) O_a and O_b are independent (or concurrent), expressed as $O_a \parallel O_b$, iff neither $O_a \rightarrow O_b$ nor $O_b \rightarrow O_a$.*

To illustrate, consider a real-time collaborative editing session with three sites, as shown in the time-space graph of Fig. 1. Three operations are generated in this scenario: operation O_1 generated at site 0 and operations O_2 and O_3 generated at site 1. An operation is first executed on the local replica, then propagated to remote sites and executed there upon their arrival. The communication channel between any pair of sites is assumed to be reliable and order-preserving. The arrows in the graph represent the propagation of operations from the local site to remote sites. Each vertical line in the graph represents the activities performed by the corresponding site. At site 1, for example, O_2 is executed first, followed by O_1 and O_3.

According to Definitions 1 and 2, there are two pairs of dependent operations in this scenario: $O_1 \rightarrow O_3$ because the execution of O_1 happens before the generation of O_3, and $O_2 \rightarrow O_3$ because the generation of O_2 happens before the generation of O_3. Moreover, there is one pair of independent operations in this scenario: $O_1 \parallel O_2$ since neither O_1 nor O_2 has been executed before the generation of the other operation.

3 The Role of Locking in Consistency Maintenance

There exist two classes of consistency in collaborative editing systems [22]: One is *syntactic* consistency, which is concerned with whether all sites have the same view of the shared document, regardless of whether the common view makes sense or not in the application context; and the other is *semantic* consistency, which is concerned with whether the common view of the shared document makes sense or not in the application context. A common misconception about locking in collaborative editing research literature is that locking can be used to maintain both classes of consistency. In this section, we explain why this is a misconception by examining the role of locking in maintaining these two different classes of consistency.

3.1 Syntactic Inconsistency Problems

In [21], three syntactic inconsistency problems—divergence, causality-violation, and intention-violation—have been identified. In this section, we will illustrate and explain why locking is not able to resolve any of them.

3.1.1 Divergence Problem

In a collaborative editing session, operations may arrive and be executed at different sites in different orders, resulting in divergent final results. As shown in Fig. 1, the three operations in this scenario are executed in the following orders: O_1, O_2, and O_3 at site 0; O_2, O_1, and O_3 at site 1; and O_2, O_3, and O_1 at site 2. If operations are not commutative (as in text editing), final editing results would not be identical among collaborating sites.

Since divergence is independent of whether or not editing operations refer to the same text region, locking is not able to resolve this problem (unless the granularity of locking is the whole document, thus prohibiting concurrency in the system). The following example is to illustrate this point.

Consider the scenario in Fig. 1. Suppose site 0 has initially placed a lock on the region covering four characters "ABCD" (which means only site 0 is allowed to delete any of these four characters and to insert any new characters in this region) and site 1 has initially placed a lock on the region covering another four characters "EFGH." Let $O_1 = Insert["1", 1]$, which is to insert the character "1" at position 1 (within the region of the lock owned by site 0); $O_2 = Delete[1, 7]$, which is to delete one character at position 7 (within the region of the lock owned by site 1); and $O_3 = Insert["3", 7]$, which is to insert the character "3" at position 7 (also within the region of the lock owned by site 1) in the document after executing both O_1 and O_2. The execution steps at each site are as follows:

At site 0: O_1 is executed first and the document becomes: "A1BCDEFGH." When O_2 arrives, it will delete the character "G" at position 7 (which is within the region of the lock owned by site 1) so the document becomes: "A1BCDEFH." When O_3 arrives and inserts "3" at position 7 (which is also within the region of the lock owned by site 1), the final document becomes:

$$"A1BCDEF3H."$$

At site 1: O_2 is executed first and the document becomes: "ABCDEFG." When O_1 arrives, it will insert "1" at position 1 (which is within the region of the lock owned by site 0) and the document becomes: "A1BCDEFG." After O_3 is generated and executed, the final document becomes:

$$"A1BCDEF3G."$$

At site 2: O_2 arrives and is executed first so the document becomes: "ABCDEFG." When O_3 arrives, it will insert "3" at position 7 (which is within the region of the lock owned by site 1) and the document becomes: "ABCDEFG3." When O_1 arrives, it will insert "1" at position 1 (which is within the region of the lock owned by site 0) and the final document becomes:

$$"A1BCDEFG3."$$

It is clear that no locking-violation has ever occurred in any step of the above execution process, i.e., every editing operation has been executed within a locking region owned by the corresponding site. Locking, however, has not prevented divergence from happening in this example—the final document states at different sites are not identical.

The divergence problem can be resolved by any serialization protocol, which ensures the final result is the same as if all operations were executed in the same order at all sites [21] or by operational transformation [5], [22], which ensures the execution of properly transformed operations in arbitrary orders produces identical results.

3.1.2 Causality Violation Problem

Since each collaborating site generates and broadcasts operations without synchronization, operations may arrive and be executed in an order different from their causal order.

As shown in Fig. 1, operation O_3 is generated after the execution of O_1 at site 1, so $O_1 \rightarrow O_3$. However, since O_3 arrives at site 2 before O_1, the execution of O_3 before O_1 may result in either an undefined operation O_3, which refers to a nonexistent context to be created by O_1, or a confused user at site 2, who observes the *effect* in O_3 before observing the *cause* in O_1.

Clearly, causality violation is only related to operation ordering and has nothing to do with whether or not operations refer to the same text region, hence locking is not able to resolve causality violation either. As reported in [21], the causality violation problem can be solved by selectively delaying the execution of some operations to enforce a causally ordered execution based on vector logical clock timestamps [18].

3.1.3 Intention Violation Problem

Due to the concurrent generation of operations, the *actual effect* of an operation *at the time of its execution* may be different from the originally *intended effect* of this operation *at the time of its generation*.

As shown in Fig. 1, operations O_1 and O_2 are generated without any knowledge of each other, so $O_1 \parallel O_2$. At site 0, O_2 is executed on a document state which has been changed by the preceding execution of O_1. Therefore, the subsequent execution of O_2 may refer to an incorrect position in the new document state and result in an editing effect different from the O_2's *intention*, which is defined as the editing effect achieved by applying O_2 on the document state from which O_2 was generated [21].

To illustrate, consider the same scenario and operations as used in the example in Section 3.1.1. After the execution of O_1 at site 0, the independent operation O_2 arrives and will delete the character at position 7, which is currently "G," rather than "H" as it was originally. Clearly, the deletion effect violates the original intention of operation O_2 (to delete "H"). As explained in Section 3.1.1, locks had been imposed on the document, but the intention violated result could not be prevented.

It should be pointed out that intention violation is an inconsistency problem of a different nature from the divergence problem. The essential difference between divergence and intention violation is that the former can always be resolved by a serialization protocol, but the latter cannot be fixed by any serialization protocol if operations were always executed in their original forms. The only known solution to intention violation is operational transformation, which adjusts an operation's parameters according to the impact of previously executed independent operations on the document state, so that the execution of the transformed operation preserves its intention.

By applying the transformation algorithms in [21] to the previous example, O_2 should be transformed against the independent operation O_1 and becomes $O_2' = Delete[1, 8]$ at site 0. The execution of O_1, O_2', and O_3 at site 0 will result in a final document state "A1BCDEF3G," which clearly preserves the intentions of all operations.

In summary, locking is unable to solve anyone of the three syntactic inconsistency problems. The only known solution to all three problems is operational transformation (integrated with a causality-preserving scheme). The reader is referred to [21] for a consistency model with properties of *Convergence*, *Causality-preservation*, and *Intention-preservation* (CCI) and an operational transformation approach to supporting this model.

3.2 Semantic Inconsistency Problems

There is another class of inconsistency problems, which could not be resolved by operational transformation, but could be resolved by locking.

To illustrate, consider a shared document with the following text:

"Transformation <u>preserve</u> operation intention."

In this text, there is an English grammar error (indicated by the underlined text), i.e., the underlined text should be "can preserve," or "preserves" or the like. Assume that two users observed the error and wanted to correct it in two different ways: One user issues an operation to insert "can" at the starting position of "preserve," while another user issues a concurrent operation to insert "s" at the ending position of "preserve." Suppose the editing system has used the operational transformation technique to ensure the CCI properties. Then, after the execution of these two independent operations at all sites, the text would be:

"Transformation <u>can preserves</u> operation intention."

From a syntactic consistency point of view, this result is correct since all sites have the same document contents and the intended effects of both operations have been achieved. This result is, however, semantically incorrect from the point of view of English grammar. In other words, operational transformation is able to ensure *plain strings* will be inserted/deleted at proper positions, even though these strings may not make a correct *English* sentence—a semantic inconsistency problem which generally cannot be automatically resolved without the intervention of human users.

However, if the users are provided with locking facilities to enforce mutual exclusion over specific regions (e.g., an English word, a statement, or a section, etc.), then either one of the two users could obtain an exclusive lock on the whole statement before modifying it and the final text would be either:

"Transformation <u>preserves</u> operation intentions"

or:

"Transformation <u>can preserve</u> operation intention,"

which apparently ensures the semantic consistency in terms of English grammar.

In general, by allowing only one user at a time to update a text region, locking can resolve semantic inconsistency problems because it prevents concurrent conflicting operations from updating the same region.

From the above discussion, it is clear that operational transformation and locking could play complementary roles in consistency maintenance in collaborative text editors: Operational transformation can be used to achieve the syntactic consistency characterized by CCI properties, whereas locking can be used to maintain data integrity by enforcing mutual exclusion over specific text regions. It was this recognition of their complementary roles that motivated our work on integrating a locking scheme with operation transformation for maintaining both syntactic and semantic consistency in collaborative editors. Moreover, since locking has no role to play for maintaining syntactic consistency and is not needed for maintaining semantic consistency when users are editing different regions of the same document most of the time, we decided to make locking optional in our system.

4 Responsive Locking and Conflict Resolution

The high responsiveness of our locking scheme is based on the notion of *tentative lock*: When a user requests a lock on an (unlocked) region, she/he is immediately granted a *tentative* lock so that she/he can edit the region without being blocked. A tentative lock will eventually become *committed* or *aborted*. When it is committed, its owner is then guaranteed to have an *exclusive* right to the region. When it is aborted, its owner will not be allowed to continue editing the region. The period between when a tentative lock is granted and when the tentative lock becomes committed/aborted is called the *transition period*.

A direct consequence of achieving high responsiveness by tentative locks is that locking conflict may occur when multiple users try to lock overlapping regions concurrently. Initially, all users are granted with tentative locks. However, the underlying locking system must ensure that only one of the tentative locks will eventually gain an exclusive updating right to the region (i.e., become committed) and the rest will be aborted. So, there is an issue of determining which tentative lock to commit at all sites in a consistent manner. In addition, at the time of locking commitment, there is an issue of how to deal with the concurrent updates on the same region made by multiple users during the transition period. In this section, two different conflict resolution protocols will be proposed for determining which concurrent locking operation will win when there is a conflict and one strategy is proposed for handling concurrent updates during the transition period.

4.1 A Distributed Protocol

The basic distributed conflict resolution protocol is defined below.

Definition 3 (Basic Distributed Conflict Resolution Protocol).

1. *When a* Lock *operation is generated at a local site:*
   a. *If the region has been locked, then the new* Lock *operation will be rejected.*
   b. *Otherwise, a tentative lock will be granted and the new* Lock *operation will be propagated to all other remote sites.*
2. *When a* Lock *operation arrives at a remote site:*
   a. *If the region is unlocked, then the new* Lock *operation will be accepted.*
   b. *If the region is locked but the new* Lock *operation has a higher priority than the existing lock, then the new operation will take over the existing lock. Otherwise, the new operation will be rejected.*

There are many ways of determining the priority of concurrent locking operations. For example, the priority can be simply defined by means of the site identifiers of locking operations: A locking operation with a smaller site identifier has a higher priority than another concurrent operation with a larger site identifier. However, this simple way of defining priority is not fair to all collaborating sites since it statically assigns higher priorities to collaborating sites with smaller site identifiers. A more fair and dynamic way of determining the priority is to use the *total ordering* "$\Rightarrow$" relationship among all operations, which can be defined by a linear logical clock [11], [20] or by a vector logical clock [21]. For any two concurrent locking operations, O_a and O_b, if $O_a \Rightarrow O_b$, then O_a has a higher priority than O_b.

Since a locking region may have an arbitrary length, the overlapping relationship among concurrent locking operations can be arbitrary as well. Moreover, concurrent locking operations may be executed in arbitrary orders at different sites. The arbitrary overlapping relationship and arbitrary execution orders of concurrent locking operations can cause inconsistency in the final locking status of the document at different sites.

To illustrate, consider three concurrent operations, $O_1 \parallel O_2 \parallel O_3$, which are trying to lock three regions with overlapping relationships as follows:

$$''\underbrace{A\,\underbrace{BCD}_{O_1}\,E\,\underbrace{FG}_{O_3}\,H}_{O_2}'',$$

which means O_1 locks "BCD," O_3 locks "FG," and O_2 locks "ABCDEFGH." Furthermore, it is assumed that $O_1 \Rightarrow O_2 \Rightarrow O_3$.

At one site, these three operations may be executed in the order of O_1, O_2, and O_3. First, O_1 is accepted. Second, O_2 is rejected since its region overlaps with O_1 and $O_1 \Rightarrow O_2$. Third, O_3 is accepted since its region does not overlap with the accepted O_1. So, the final locking status of the document at this site should look like:

$$"A\underbrace{BCD}_{O_1}E\underbrace{FG}_{O_3}H."$$

At another site, however, these three operations may be executed in a different order: O_2, O_3, and O_1. First, O_2 is accepted. Second, O_3 is rejected since its region overlaps with O_2 and $O_2 \Rightarrow O_3$. Third, O_1 takes over O_2 since its region overlaps with O_2 and $O_1 \Rightarrow O_2$. In this case, the final locking status of the document at this site would look like:

$$"A\underbrace{BCD}_{O_1}EFGH."$$

Clearly, the final locking status at these two sites is inconsistent. One way of resolving this inconsistency is to ensure that the final locking status of the document at all sites will be the same as if all concurrent operations were executed in the order of their priority, regardless of their arbitrary execution orders at different sites. This locking effect is called the *serialized locking effect*. To achieve the serialized locking effect, we need to keep track of all executed locking operations, accepted, or rejected in the internal locking table *LT* at each site. When a remote *Lock* operation arrives at a site, it is checked against accepted operations in the *LT* with a higher priority than the new operation to see whether the new operation is overlapping with any of them. If yes, the new operation is rejected (but still recorded in the *LT*). Otherwise, the new operation is accepted and all operations in the *LT* with a lower priority will be reassessed for acceptance or rejection. More precisely, the extended distributed conflict resolution protocol is defined below.

Definition 4 (Extended Distributed Conflict Resolution Protocol).

1. *When a* Lock *operation is generated at a local site:*
   a. *If the region has been locked, then the new* Lock *operation will be rejected.*
   b. *Otherwise, a tentative lock will be granted and the new* Lock *operation will be propagated to all other remote sites.*
2. *When a* Lock *operation arrives at a remote site:*
   a. *If the region is locked by an operation with a higher priority than the new operation, then the new operation is rejected, but recorded in the LT.*
   b. *Otherwise, the new operation is accepted and all operations in the LT with a lower priority than the new operation are reassessed for acceptance or rejection one by one in the order of their priority, taking into account the effect of the new operation.*

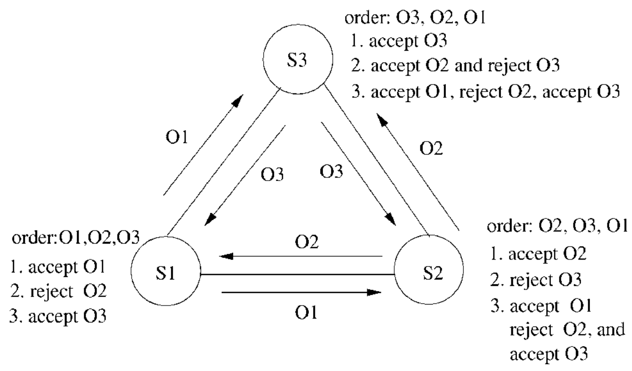

Fig. 2. A scenario of distributed conflict resolution protocol.

To illustrate how the extended distributed conflict resolution protocol works, consider a system with three fully connected collaborating sites, as shown in Fig. 2. Suppose three concurrent locking operations, O_1, O_2, and O_3, are generated by the three sites, respectively. Assume $O_1 \Rightarrow O_2 \Rightarrow O_3$ and O_2 overlaps with both O_1 and O_3, but O_1 and O_3 do not overlap, which is similar to the previous example.

At site 1, the execution order is: O_1, O_2, and O_3. A tentative lock for O_1 is first granted and then O_1 is propagated to sites 2 and 3. When O_2 arrives, it will be rejected since its region overlaps with O_1 and $O_1 \Rightarrow O_2$. When O_3 arrives, it will be accepted since its region does not overlap with the accepted O_1.

At site 2, the execution order is: O_2, O_3, and O_1. A tentative lock for O_2 is first granted and then O_2 is propagated to sites 1 and 3. When O_3 arrives, it will be rejected since its region overlaps with O_2 and $O_2 \Rightarrow O_3$, but O_3 is still recorded in the *LT*. When O_1 arrives, it will be accepted since no operation with a higher priority overlaps with O_1. Then, operations with lower priorities—the accepted O_2 and the rejected O_3—in the *LT* will be reassessed in the presence of O_1: O_2 will be rejected since its region overlaps with O_1 and O_3 will be accepted since its region does not overlap with the accepted O_1.

At site 3, the execution order is: O_3, O_2, and O_1. A tentative lock for O_3 is first granted and then O_3 is propagated to sites 1 and 2. When O_2 arrives, it will take over O_3 since it overlaps with O_3 and $O_2 \Rightarrow O_3$. When O_1 arrives, it will be accepted since its region does not overlap with any operation with a higher priority. Then, operations with lower priorities—the accepted O_2 and the rejected O_3—in the *LT* will be reassessed in the presence of O_1: O_2 will be rejected since its region overlaps with O_1 and O_3 will be accepted since its region does not overlap with the accepted O_1.

After executing three operations at all sites, the final locking status of the document at all sites is consistent: O_1 and O_3 were accepted and O_2 was rejected.

There are two important points related to the distributed protocol which should be pointed out explicitly. First, a tentative lock may be temporarily rejected during its transition period. For example, at site 3 in Fig. 2, O_3 was first accepted, then rejected due to the acceptance of O_2, and finally reaccepted due to the acceptance of O_1 and the rejection of O_2. Although this *accept-reject-accept* phenomena does not have impact on the final locking status of the document, it does have an impact on the user interface since the user issuing O_3 may temporarily be prohibited from updating the region associated with O_3, which may cause confusion to this user. Second, if there were one additional collaborating site x in Fig. 2 and if this site did not generate any operation, after executing the three operations at all sites, all sites (except site x) would not be able to know whether the locks imposed by O_1 and O_3 had been committed since they did not know whether site x had generated any concurrent and overlapping operation with a higher priority. Only after these sites had received a status updating message from site x could they be confirmed as to the commitment of locks imposed by O_1 and O_3. If speedy confirmation of the commitment/abortion of a tentative lock is needed, the distributed conflict resolution protocol has to require every site to broadcast an acknowledgment message after receiving a locking operation (if this site has not broadcast any operation which is concurrent with the received locking operation).

For the system with three sites in Fig. 2, two messages are needed for processing each locking operation. In general, for a system of N collaborating sites, the distributed conflict resolution protocol needs $N-1$ messages for processing a locking operation without speedy confirmation of commitment or abortion or $(N-1) \times (N-1)$ messages (in the worst case) for processing a locking operation with speedy confirmation of commitment or abortion. The (worst case) length of transition period for a tentative lock to become committed or aborted equals the round-trip time for a locking operation to travel from its generating site to the most distant remote site and vice versa.

4.2 A Coordinator-Based Protocol

Alternatively, locking conflict resolution can be based on the use of a centralized *locking coordinator*, which may be a dedicated site or a site elected from one of the N collaborating sites. The coordinator-based conflict resolution protocol is defined below.

Definition 5 (Coordinator-Based Conflict Resolution Protocol).

1. *When a* Lock *operation is generated at a local site:*
   a. *If the region has been locked, then the new* Lock *operation will be rejected.*
   b. *Otherwise, a tentative lock will be granted and the new* Lock *operation will be sent to the coordinator site.*
2. *When a* Lock *operation from a collaborating site arrives at the coordinator site:*
   a. *If the region has been locked, then the remote operation will be rejected.*
   b. *Otherwise, the remote operation will be accepted and broadcast to all collaborating sites (including the site from which the operation came).*
3. *When a* Lock *operation from the coordinator site arrives at a collaborating site, it is always accepted as a committed lock. Moreover, any operation in the LT which is overlapping with the newly committed lock should be removed.*

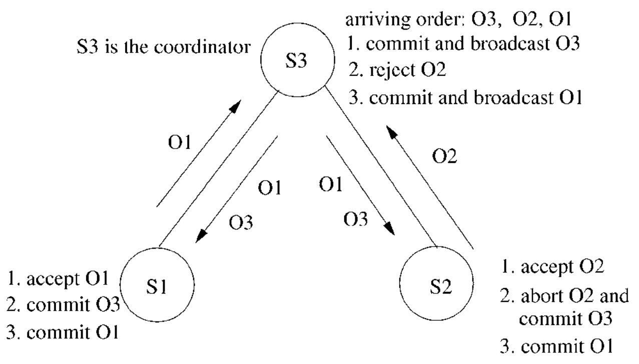

Fig. 3. A scenario for coordinator-based conflict resolution protocol.

Under the coordinator-based protocol, when multiple concurrent and overlapping locking operations are generated, the operation first arriving the coordinator will win. In other words, the final locking effect for a group of arbitrarily overlapping and concurrent locking operations is defined by the order in which they arrive at the coordinator.

To illustrate how the coordinator-based conflict resolution protocol works, consider a system with three collaborating sites, with site 3 being the locking coordinator, as shown in Fig. 3. Suppose three concurrent locking operations, O_1, O_2, and O_3, are generated by these three sites, respectively. Assume O_2 overlaps with both O_1 and O_3, but O_1 and O_3 do not overlap, which is similar to the scenario in Fig. 2. Moreover, it is assumed that these three operations arrive at the coordinator in the order of O_3, O_2, and O_1.

At the coordinator site, O_3 is local and will be the first one accepted and broadcast to sites 1 and 2. When O_2 arrives, it will be rejected and not broadcast to any site since it overlaps with the committed O_3. When O_1 arrives, it will be accepted, committed, and broadcast to sites 1 and 2 since it does not overlap with the committed O_3.

At site 1, a tentative lock for O_1 is first granted and then O_1 is sent to the coordinator. When O_3 arrives at site 1, it will be simply accepted and committed. When O_1 arrives at site 1, the tentative lock imposed by O_1 will become committed.

At site 2, a tentative lock for O_2 is first granted and then O_2 is sent to the coordinator. When O_3 arrives, it will be accepted as a committed lock. Moreover, the tentative lock imposed by O_2 will be aborted since it overlaps with the committed lock imposed by O_3. When O_1 arrives, it will simply be accepted and committed.

At the end of this scenario, all three sites have the same two locks imposed by O_1 and O_3. In addition, all sites have been confirmed as to the commitment of these two locks. In this scenario, three messages are used for processing and confirming a committed lock and one message is used for processing an aborted lock (O_2). There is no need for additional acknowledge messages for confirming locking commitment or abortion.

In general, for a system of N sites, the coordinator-based conflict resolution protocol needs N messages for processing and confirming a committed lock and only one message for processing an aborted lock. The (worst-case) length of transition period for a tentative lock to become committed or aborted is equal to the round-trip time for a locking operation to travel from its generating site to the coordinator site and vice versa.

Compared to the distributed protocol, the coordinator-based protocol needs a smaller total number of messages for processing and confirming each locking operation (N versus $(N-1) \times (N-1)$). The coordinator needs to receive and broadcast almost the same number of locking operations as a normal collaborating site would do in the distributed protocol, but the coordinator uses a simpler algorithm to check the acceptance or rejection of a new locking operation. So, the work load performed by the coordinator is less than a normal collaborating site in the distributed protocol. Finally, a tentative lock stays in the accepted mode until it becomes committed or aborted in the coordinator-based protocol, whereas a tentative lock may go through an *accept-reject-accept* process in the distributed protocol. Because of the above considerations and because of the availability of an existing central server in our web-based collaborative environment, the coordinator-based protocol has been adopted and implemented in our web-based prototype system.

4.3 Strategies for Handling Concurrent Updates

During the transition period, there may exist multiple tentative locks on the same region and multiple users may update the same region concurrently. At the time of committing a tentative lock, if the region has been updated by multiple users, which user's updates should be retained and which one's should be undone?

One possibility is to undo all concurrent updates made by all users during the transition period. The problem with this strategy is that all work done during the transition period gets lost due to concurrency, which is too conservative since concurrency does not necessarily cause inconsistency.

Another possibility is to retain the updates made by the user whose tentative lock has become committed and to undo concurrent updates made by other users whose tentative locks have been aborted, which is the strategy taken by some existing optimistic locking schemes [7], [8]. The problem with this strategy is that some users' work gets lost due to concurrent updates and users cannot see what other users have done during the transition period, thus losing important hints about users' intentions, which can be useful in deciding what to do next in the collaboration process.

The final possibility is to retain all concurrent updates made by all the users during the transition period and to give the user with a committed lock the choice to undo or retain the updates by other users. This is the strategy taken by our system. The arguments for this strategy are:

1. It ensures the work done by a user is never distroyed by concurrent operations from other users, which is one important requirement for *intention preservation* [21] and
2. It provides users with a complete picture about what others intended to do so that they can make a better assessment of the situation and decide what to do next.

For example, the user with a committed lock may decide to use the system-provided undo facility [24] to undo any update performed during the transition period or to keep all updates if that is desired.

In general, we advocate a design principle for collaborative systems: In case of conflict (caused by concurrency), it is usually better to preserve all users' work to facilitate a user-decided solution to the conflict, rather than to destroy some users' work to impose a system-decided solution to the conflict. Conflicts among collaborative users are better understood and resolved by these users if the system can provide explicit information about all users' actions and suitable undo facilities to support error recovery.

5 Consistent Fine-Grain Locking by Operational Transformation

Fine-grain locking operations are similar to editing operations in their way of representing a locking/editing region by means of its starting position and its length (see Section 2.2). Similar syntactic inconsistency problems which occur to editing operation, may also occur to locking operations. Therefore, there is an issue of consistency maintenance for locking operations in the system. In this section, we will show how operational transformation can be extended to support consistent fine-grain locking.

To illustrate inconsistency problems and solutions under the circumstance of mixed locking and editing operations, we assume that editing operations are transformed before their executions, but locking operations are always executed in their original forms.

5.1 Problems and Solutions

5.1.1 Preserving Locking Causality

When locking operations are mixed with editing operations in the system, their causal ordering must also be respected. Otherwise, problems may occur.

To illustrate, consider the scenario in Fig. 1. Let

$$O_1 = Insert["1234," 0],$$

$O_2 = Delete[4,4]$ (to delete "EFGH"), and $O_3 = Lock[8,0]$ (to lock "1234ABCD"). At site 2, after executing O_2, the document state becomes:"EFGH." Since O_3 arrives at site 2 before O_1, O_3 will try to lock eight characters in the document which has only four characters at the moment!

Clearly, this problem has the same symptom as the causality violation problem for editing operations and, hence, can be resolved by using the same causality preserving scheme based on vector logical clocks [3], [18]. Therefore, the solution is to timestamp locking operations by vector logical clocks and to execute locking operations only when they are causally ready. For this particular example, the properly timestamped O_3 will be suspended until O_1 has arrived and been executed at site 2.

5.1.2 Preserving Locking Intentions

When a locking operation is generated concurrently with editing operations, the actual effect of this locking operation at the time of its execution may not be the same as the intended effect at the time of its generation.

To illustrate, consider again the scenario in Fig. 1. Let $O_1 = Lock[6, 1]$, which intends to lock the region beginning at position 1 and covers six characters "BCDEFG," as indicated below:[1]

$$''A\underbrace{BCDEFG}_{R[s_0,6,1]}H'';$$

$O_2 = Insert[''123,''0]$, which is to insert three characters "123" at position 0; and $O_3 = Delete[1, 2]$, which is to delete one character "3" at position 2. At site 1, O_2 is executed first, and the document becomes: "$123ABCDEFGH$." Then, O_1 arrives and is executed in its original form. So, the document locking status becomes:

$$''1\underbrace{23ABCD}_{R[s_0,6,1]}EFGH,''$$

which obviously violates the intention of O_1, which is to lock the region covering "BCDEFG," rather than "23ABCD." After that, when O_3 is generated to delete character "3" at position 2, it will fail because character "3" is currently being locked by site 0!

Clearly, this problem has the same symptom as the intention-violation problem for editing operations and, hence, can also be resolved by the same operational transformation technique. Therefore, the solution is to apply transformation on locking operations before executing them. For this particular example, when the locking operation O_1 arrives, it should be transformed against the concurrent editing operation O_2 and becomes $O'_1 = Lock[6, 4]$ (see Section 5.2.1 for the definition of the transformation function). The execution of O'_1 will result in the following document locking status:

$$''123A\underbrace{BCDEFG}_{R[s_0,6,4]}H,''$$

which preserves the intention of O_1.

5.1.3 Dealing with Concurrent and Overlapping Locking and Editing Operations

Since locking is optional, an editing operation may fall into the locking region of a concurrent locking operation. For example, consider the scenario in Fig. 1. Let $O_1 = Lock[6, 1]$, and $O_2 = Insert[''123,''2]$. Clearly, the insertion position of O_2 falls in the locking region covered by O_1. Then, after executing both O_1 and O_2, what is the document content and locking status?

One possible scheme of dealing with this situation is to treat locking operations as more privileged than editing operations by prohibiting (or undoing) the execution of concurrent overlapping editing operations. In the above example, O_2 will either be prohibited from being executed at site 0 or be undone at site 1 and 2. In this scheme, an editing operation may fail due to its region overlapping with a concurrent locking operation.

1. The brace under a text region represents a locked region labeled by $R[s_i, N, P]$, where s_i is the identifier of the site which owns this region, N is the length of the region, and P is the starting position of the region.

Another possible scheme is to treat concurrent locking and editing operations equally by accommodating the effects of both: The locking region will expand or shrink to accommodate the effects of concurrent editing operations. In the above example, after executing both O_1 and O_2, the locking region of O_1 will be expanded to cover the characters inserted by O_2 and the document content and locking status will become:

$$''A\underbrace{B123CDEFG}_{R[s_0,9,1]}H.''$$

Following the general principle that when there is a conflict (due to concurrency), it is better to preserve the effects of all operations (see Section 4.3), the second scheme has been chosen for our system. One additional advantage of choosing the second scheme is that it can be supported by using the same operation transformation technique (see Section 5.2) for supporting intention preservation.

5.1.4 Adjusting Locking Regions after Editing

A locking region may be changed not only by concurrent editing operations, but also by subsequent editing operations under the following circumstances:

1. When a subsequent insertion or deletion operation is performed within a locked region, the locked region will expand or shrink.
2. When a subsequent insertion or deletion operation is performed on the left side of a locked region, the locked region may shift to right or left.

Therefore, a *Locking Region Adjustment* (LRA) scheme is needed to adjust existing locking regions after executing every editing operation.

For example, suppose the current document locking status is as follows:

$$''A\underbrace{BCDEF}_{R[s_0,5,1]}GH\underbrace{IJKLMNO}_{R[s_1,7,8]}PR\underbrace{STUVWX}_{R[s_2,6,17]}YZ,''$$

where three users have locked three different regions of the document. Suppose the user at site 1 issues a new editing operation $Insert[''12'', 11]$ to insert string "12" at position 11, which is within the region locked by this user. After executing this editing operation, the locking region owned by site 1 should expand to cover the two new characters, and the locking region owned by site 2 should be shifted to the right by two positions, but the locking region owned by site 0 remains unchanged, as shown below:

$$''A\underbrace{BCDEF}_{R[s_0,5,1]}GH\underbrace{IJK12LMNO}_{R[s_1,9,8]}PR\underbrace{STUVWX}_{R[s_2,6,19]}YZ.''$$

The adjustment of a locking region with respect to a subsequent editing operation is essentially the same as the transformation of a locking operation against an editing operation. Therefore, the LRA scheme in our system will be realized by using the same functions devised for locking operation transformation (to be described in Section 5.2.1).

Moreover, locking regions may also need adjustment after executing a *Lock* operation that overlaps with existing locked regions belonging to the same user. When this happens, existing locked regions may be merged into a bigger new region.

For example, suppose the current document locking status is as follows:

$$''A\underbrace{BCDEF}_{R[s_1,5,1]}GH\underbrace{IJKLMNO}_{R[s_1,7,8]}PR\underbrace{STUVWX}_{R[s_2,6,17]}YZ.''$$

Suppose the user at site 1 issues a new operation $Lock[11, 0]$ to lock a region starting from position 0 and covering 11 characters. Clearly, the locking region of this new operation overlaps with two existing locking regions owned by the same user. Therefore, after executing this new locking operation, the two existing regions will be merged into one bigger region and the locking status of the document becomes:

$$''\underbrace{ABCDEFGHIJKLMNO}_{R[s_1,15,0]}PR\underbrace{STUVWX}_{R[s_2,6,17]}YZ.''$$

To unlock a locked region, a user only needs to place her/his curser at any position in that region and then click the unlock button at the interface. The region of an *Unlock* operation is always equal to an existing locked region.

5.2 Locking Operation Transformation

In this section, we discuss the extension of operational transformation to support consistent fine-grain locking.

5.2.1 Transformation Functions

Types of transformation functions. To support editing operations transformation, two types of transformation functions—*inclusion* and *exclusion*—have been defined in [21]. The inclusion transformation (*IT*) function transforms operation O_a against operation O_b in such a way that the impact of O_b is effectively included. The exclusion transformation (*ET*) function transforms O_a against O_b in such a way that the impact of O_b is effectively excluded. The ET and IT functions are *reversible* in the sense that if $O'_a = IT(O_a, O_b)$, then it must be that $O_a = ET(O'_a, O_b)$.

Since there is never a need to apply exclusion transformations on a locking operation against an editing operation in the *HB*, we only need to define inclusion transformation functions between locking and editing operations. Moreover, since executed locking operations are not saved in the *HB* and do not change the contents of the document, there is never a need to transform an editing operation against a locking operation. Therefore, we only need to define functions for transforming locking operations against editing operations. Finally, since the transformation strategies for both *Lock* and *Unlock* operations are essentially the same, the following discussion will be confined to *Lock* operation transformation functions only.

To facilitate the description of transformation functions, the following notations are introduced:

1. $P(O)$: the position parameter of operation O.
2. $L(O)$: the length of operation O. For *Insert*, it is the the number of characters to be inserted. For *Delete*, it is the number of characters to be deleted. For *Lock*, it is the number of characters to be locked.

Basic transformation strategy. To transform locking operation O_a against editing operation O_b, the inclusion transformation function applies the following transformation strategy:

1. compare the parameters of O_a and O_b to determine the relationship of their operation regions;
2. assume that O_b has been executed to find O_a's new locking region in the changed document state; and
3. adjust O_a's parameters to produce a new operation O'_a, according to the comparison result in Step 1 and the impact of the assumed execution of O_b in Step 2.

Transforming *Lock* **against** *Insert*. $IT\_LI(O_a, O_b)$ is defined to apply the inclusion transformation to a *Lock* operation O_a against an *Insert* operation O_b.

```
Function 1 $IT\_LI(O_a, O_b)$
  {
    if $P(O_b) \geq P(O_a) + L(O_a)$ $O'_a := O_a$;
    else if $P(O_a) > P(O_b)$
        $O'_a := Lock[L(O_a), P(O_a) + L(O_b)]$;
    else $O'_a := Lock[L(O_a) + L(O_b), P(O_a)]$;
    return $O'_a$;
  }
```

If O_b refers to a position which is to the *right* of the locking region covered by O_a (i.e., $P(O_b) \geq P(O_a) + L(O_a)$), then the execution of O_b must not have any impact on the parameters of O_a. Therefore, no adjustment is made to O_a. However, if O_b refers to a position which is to the *left* of the locking region covered by O_a (i.e., $P(O_a) > P(O_b)$), then the execution of O_b must shift the locking region of O_a to the *right*. Therefore, the position parameter of O_a is incremented by $L(O_b)$. The last case is that the insertion position of O_b falls into the locking region covered by O_a, so the length parameter of O_a is incremented by $L(O_b)$.

For example, suppose the initial document contains the string "ABCDEF" and there are two concurrent operations $O_a = Lock[5, 1]$ (issued by site 0 to lock "BCDEF") and $O_b = Insert[''123,'' 0]$. If O_b is executed before O_a, O_a should be transformed against O_b. The transformation outcome will be $O'_a = IT\_LI(O_a, O_b) = Lock[5, 4]$ since $(P(O_a) = 1) > (P(O_b) = 0)$. After executing O_b and O'_a, the document state and locking status would be:

$$''123A\underbrace{BCDEF}_{R[s_0,5,4]}.''$$

Transforming *Lock* **against** *Delete*. $IT\_LD(O_a, O_b)$ is defined to apply the inclusion transformation to a *Lock* operation O_a against a *Delete* operation O_b.

```
Function 2 $IT\_LD(O_a, O_b)$
  {
    if $P(O_b) \geq (P(O_a) + L(O_a))$ $O'_a := O_a$;
    else if $P(O_a) \geq (P(O_b) + L(O_b))$
        $O'_a := Lock[L(O_a), P(O_a) - L(O_b)]$;
    else if $P(O_b) \leq P(O_a)$ and
            $(P(O_a) + L(O_a)) \leq (P(O_b) + L(O_b))$
        $O'_a := Lock[0, P(O_b)]$;
    else if $P(O_b) \leq P(O_a)$ and
            $(P(O_a) + L(O_a)) > (P(O_b) + L(O_b))$
        $O'_a := Lock[P(O_a) + L(O_a) -$
                    $(P(O_b) + L(O_b)), P(O_b)]$;
    **else if** $P(O_b) > P(O_a)$ **and**
            $(P(O_b) + L(O_b)) \geq (P(O_a) + L(O_a))$
        $O'_a := Lock[P(O_b) - P(O_a), P(O_a)]$;
    **else** $O'_a := Lock[L(O_a) - L(O_b), P(O_a)]$;
    **return** $O'_a$;
}
```

If O_b refers to a position which is to the *right* of the locking region covered by O_a (i.e., $P(O_b) \geq P(O_a) + L(O_a)$), then the execution of O_b must not have any impact on the parameters of O_a. Therefore, no adjustment is made to O_a. If O_b refers to a position which is to the *left* of the locking region covered by O_a (i.e., $P(O_a) \geq P(O_b)$), then the execution of O_b must shift the locking region of O_a to the *left*. Therefore, the position parameter of O_a is decremented by $L(O_b)$. If the locking region of O_a overlaps with the deleting region of O_b (i.e., the last four cases in $IT\_LD(O_a, O_b)$), the execution of O_b must remove the overlapping region from the locking region of O_a. For the special case that the locking region of O_a is completely covered by the deleting region of O_b (i.e., $P(O_b) \leq P(O_a)$ and $(P(O_a) + L(O_a)) \leq (P(O_b) + L(O_b))$), the length of the locking region of O_a will be reduced to *zero*.[2]

For example, suppose the initial document contains the string "ABCDEF" and there are two concurrent operations $O_a = Lock[6, 0]$ (issued by site 0 to lock all six characters) and $O_b = Delete[2, 4]$ (to delete "EF"). If O_b is executed before O_a, O_a should be transformed against O_b. The transformation outcome will be $O'_a = IT\_LD(O_a, O_b) = Lock[4, 0]$ since $P(O_b) > P(O_a)$ and $(P(O_b) + L(O_b)) \geq (P(O_a) + L(O_a))$. After executing O_b and O'_a in sequence, the document state and locking status would be:

$$"\underbrace{ABCD}_{R[s_0,4,0]}."$$

5.2.2 Transformation Control Algorithm

Under two circumstances, a locking operation needs to be transformed against an executed editing operation. First, when an editing operation has been executed, each locking operation in the *LT* needs to be transformed against this newly executed editing operation. Under this circumstance, the IT_LI and IT_LD functions can be directly applied between the locking operation and the new editing operation. Second, when a remote locking operation becomes causally ready for execution, it needs to be transformed against executed concurrent editing operations in the *HB*. In this case, simply applying the IT_LI and IT_LD functions between the new locking operation and concurrent editing operations in the *HB* may not always produce the correct result due to the fact that concurrent editing and locking operations may be generated and defined on different document states.

2. It should be noted that reducing a locking region to a zero length does not remove the lock from the *LT*. The owner of this zero-length lock may insert a new string at the lock's position, thus extending its locking region. A zero-length lock may also be generated directly from the user interface if the user wants to protect a string being inserted at a particular position in the document.

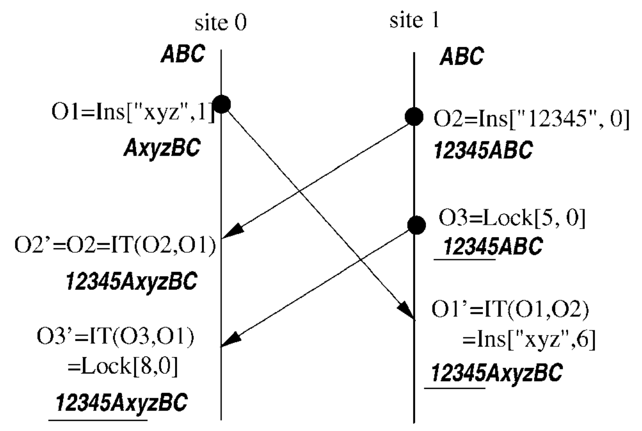

Fig. 4. A scenario for illustrating incomparable concurrent operations. The underline for a string of characters represents a locking region.

To illustrate the problem, consider the scenario in Fig. 4: The initial document contains the string "ABC"; site 0 generates operation $O_1 = Insert["xyz", 1]$; and site 1 independently generates operations $O_2 = Insert["12345", 0]$ and $O_3 = Lock[5, 0]$ in sequence. After executing these three operations, the document content and locking status at both sites should be:

$$"\underbrace{1\,2\,3\,4\,5}_{R[s_1,5,0]}AxyzBC,"$$

which preserves the intentions of all three operations.

At site 1, O_2 and O_3 are executed in sequence. When O_1 arrives, it is transformed against the concurrent editing operation O_2 to become $O'_1 = Insert["xyz," 6]$ since the insertion position of O_1 was at the right side of the insertion position of O_2. After the execution of all three operations, site 1 has the correct final document state and locking status, as shown in Fig. 4.

At site 0, however, the situation is different. After the execution of O_1, the document becomes: "AxyzBC." When O_2 arrives, it is transformed against the concurrent operation O_1 to become $O'_2 = O_2$ since the insertion position of O_2 was on the left side of the insertion position of O_1. After executing O_2, the document becomes "12345AxyzBC." Finally, when the locking operation O_3 arrives, it will be transformed against the concurrent editing operation O_1. By comparing the locking region of O_3 (to lock the region from position "0" to position "5") with the insertion position of O_1 (to insert at the position "1"), it seems that O_1 is inserting in the locking region of O_3 (but this is actually wrong). According to our scheme for handling concurrent and overlapping locking and editing operations (see Section 5.1.3), the locking region of O_3 should be extended by a length of "3" to accommodate the three characters inserted by O_1. Therefore, the transformed O_3 will be $O'_3 = Lock[8, 0]$. After executing O'_3, the document content and locking status at site 0 look will be:

$$"\underbrace{12345Axy}_{R[s_1,8,0]}zBC,"$$

which is obviously incorrect!

The root of the problem is that the two concurrent operations O_1 and O_3 were not generated (and, hence, not defined) from the same document state: O_1 was generated when the document state was "ABC," but O_3 was generated when the document state was "12345ABC" after the execution of O_2. Consequently, the parameters in these two operations are not directly comparable for assessing their overlapping relationship and the *IT* function would not be able to produce the right result.

This is a nontrivial problem. The main challenge here is how to ensure that the two input operations will be defined on the same document state in every application of the *IT* function. Fortunately, we found that this problem is similar to the transformation control problem for pure editing operations and, hence, solutions devised for controlling editing operation transformation can be applied to locking operations as well. In the following, the generic operational transformation control algorithm GOTO proposed in [22] will be directly applied for controlling locking operation transformation.

The GOTO algorithm will transform a new locking operation *O* according to the following three different situations in the *HB*: First, if there is no executed editing operation in the *HB* which is concurrent with *O*, then *O* can be executed without transformation. Second, if there are some operations in the *HB* which are concurrent with *O* and they are all placed after the operations causally before *O* in the *HB*, then *O* is transformed against these concurrent operations one by one to determine its execution form. Third, if operations concurrent with or causally before *O* are mixed in their orders in the *HB*, then all operations in the *HB* will be reordered (by transformation and position shift) in such a way that operations which are causally before *O* are positioned on the left side of the *HB* and operations which are concurrent with *O* are positioned on the right side. Then, *O* is transformed against concurrent operations one by one to determine its execution form.

To describe the GOTO algorithm precisely, the function $LIT(O, L)$ is defined to apply the *IT* function on operation *O* against a list of operations in *L*.

```
LIT(O, L) {
  for(i = 0, i < sizeof(L); i++)
   O = IT(O, L[i]);
  return O;
}
```

Moreover, a procedure `LTranspose(L)` is defined below to transform and circularly shift the list of operations in *L*.

```
LTranspose(L) {
  for(i = sizeof(L) - 1; i > 1, i-) {
        O = ET(L[i], L[i-1]); L[i] = IT(L[i-1],
O);
        L[i]=IT(L[i-1], O);
     L[i-1] = O;
  }
}
```

Algorithm 1 $GOTO(O, L) : O'$. O: a new locking operation. L: the list of editing operations $[O_0, O_1, \ldots, O_m]$ in the HB. O': the transformed form of O.

1. Scan $L[0, m]$ from left to right to find the first operation O_k such that $O_k \parallel O$. If no such an operation is found, then **return** $O' := O$.
2. Otherwise, scan $L[k+1, m]$ to find operations causally before O. If no single such operation is found, then return$O' := LIT(O, L[k, m])$.
3. Otherwise, let $L_1 = [O_{c_1}, \ldots, O_{c_r}]$ be the list of operations in $L[k, m]$ which are causally before O.
   - $LTranspose(L[k+i-1, c_i])$, for $1 \le i \le r$;
   - **return** $O' := LIT(O, L[k+r, m])$.

To illustrate how the GOTO algorithm works, we apply the GOTO algorithm to the processing of O_3 at site 0 in Fig. 4. When O_3 arrives at site 0, $GOTO(O_3, L = [O_1, O_2])$ is invoked to determine the execution form of O_3 as follows:

1. Scan $[O_1, O_2]$ to find the first operation O_1 which is concurrent with O_3.
2. Scan $[O_2]$ to find the first operation O_2 which is causally before O_3.
3. Apply $LTranspose(L = [O_1, O_2])$ to transform and swap O_1 and O_2 as follows:
   - $L[0] := O_2 = ET(O_2, O_1)$ since
   $$O_2 = IT(O_2, O_1),$$
   i.e., the execution of O_1 had no impact on the original parameters of O_2.
   - $L[1] := O'_1 = Insert["xyz", 7] = IT(O_1, O_2)$, where O'_1 is the form that O_1 would have taken if it was generated after the execution of O_2.
   - Return $O'_3 := O_3 = IT(O_3, O'_1)$, which is the correct execution form of O_3 (see the definition of *IT_LI*).

From this example, we can also see that, in each of the two applications of the *IT* function, the pair of input operations are indeed defined on the same document state: 1) O_1 and O_2 in $IT(O_1, O_2)$ are defined on the same document state "ABC"; and 2) O_3 and O'_1 in $IT(O_3, O'_1)$ are defined on the same document state "12345ABC."

6 Implementation Algorithms

In this section, we discuss how to implement responsive and consistent fine-grain locking based on the solutions devised in the previous sections.

6.1 The Locking Table

The major data structure for supporting fine-grain locking is the locking table *LT* maintained by each site. Each entry in the *LT* is a *Lock* operation which represents a locked region belonging to a single user. A successful execution of a *Lock* operation adds an entry into the *LT* or merges several entries belonging to the same user into one bigger entry. A successful execution of an *Unlock* operation may remove an existing entry from the *LT*. When a site starts a new session, its *LT* is initialized to an empty table. When a site joins an existing session, its *LT* is initialized to the concurrent content of an existing site. When a site leaves a session, all locks owned by this site must be removed from the *LT*s at other sites. No entry can have an overlapping region with another entry in the *LT* for ensuring exclusiveness. When the system is in a quiescent status (i.e., all sites have executed the same collection of operations and there is no operation in transit), the *LT*s have the same content at all sites. A successful execution of a *Lock* operation adds an entry into the *LT* or merges several entries belonging to the same user into one bigger entry. A successful execution of an *Unlock* operation may remove an existing entry from the *LT*. No entry can have an overlapping region with another entry in the *LT* for ensuring exclusiveness.

6.2 Local Operation Permission Check

With optional locking in place, a locking or editing operation can be executed if and only if the operation region is either unlocked or locked by the same user. When an operation is generated at a local site, it needs to pass a *Local Permission Check* (LPC), as defined below. In the following discussion, the notation *Sid*(*O*) is used to denote the site identifier of operation *O*.

```
LPC(O, LT) {
  for(i = 0; i<sizeof(LT); i++) {
    if(Sid(O)!=Sid(LT[i])) &&
      overlap(O,LT[i]))
      return false;
  }
  return true;
}
```

The function `LPC(O, LT)` scans the local locking table *LT* to check whether there exists any entry belonging to a different user (i.e., `Sid(O) != Sid(LT[i])`) and overlapping with the newly generated operation *O* (by invoking `overlap(O, LT[i])`). If one such entry is found, *O* is not permitted and *false* is returned; otherwise, *O* is permitted and *true* is returned.

If the permission check is successful, the operation will be executed immediately at the local site. If *O* is an editing operation, it will update the document state and then be put into the local *HB*. If *O* is a *Lock* operation, it will be put into the local *LT*, which means a tentative lock has been granted to the local user. If *O* is an *Unlock* operation, it will remove the corresponding locking entry from the local *LT*. A locking operation is regarded as having been executed at a site if and only if this operation has finished its manipulation of the *LT* at that site.

It should be stressed that the process of checking and executing a local operation is nonblocking. Only the local *LT* needs to be examined for permission check, the local *HB* and document state are updated for executing an editing operation, and the local *LT* is updated for executing a locking operation. There is never a need for consulting and communicating with any remote site, so the responsiveness of local operations is good. After the local operation has been executed, it is timestamped and propagated to remote sites.

6.3 Remote Operation Processing

When an operation arrives at a remote site and becomes causally ready for execution, it is transformed against the editing operations in the *HB* under the control of the GOTO algorithm.

If the remote operation is an editing operation, it is guaranteed to have permission for execution at the remote site according to the scheme for handling concurrent and overlapping editing and locking operations (see Section 5.1.3).

For remote locking operation processing, we assume the coordinator-based conflict resolution protocol (see Section 4.2) is adopted. When a remote locking operation arrives at the coordinator site, if it is an *Unlock* operation, it is executed by simply removing the corresponding entry from the *LT* at the coordinator site and then it is broadcast to all other sites except the one from which the operation came. If the remote locking operation is a *Lock* operation, the following *Coordinator-based Conflict Resolution (CCR)* procedure will be executed.

```
CCR(O, LT) {
  for(int i = 0; i<sizeof(LT); i++)
    if(Sid(O)!=Sid(LT[i])) &&
        overlap(O,LT[i]))
      return;  // reject O
  putinLT(O, LT);// accept O
  broadcast(O);
  return;
}
```

The `CCR (O, LT)` procedure scans the locking table *LT* at the coordinator site to check whether there is an entry which belongs to a different user (i.e., `Sid(O)!=Sid(LT[i])`) and overlaps with *O* (by invoking `overlap(O,LT[i])`). If one such entry is found, *O* is rejected since another concurrent locking operation has been executed at the coordinator site. Otherwise, *O* is accepted and put in the *LT* (by `putinLT(O,LT)`)[3] and then *O* is broadcast to all remote sites (by `broadcast(O)`) to notify the commitment of its locking status.

When a remote locking operation from the coordinator arrives at a noncoordinator site, if it is an *Unlock* operation, it is executed by simply removing the corresponding entry from the *LT* at this site. If the remote locking operation is a *Lock* operation, the following *Committed Lock Processing (CLP)* procedure will be executed:

```
CLP(O, LT) {
  for(int i = 0;i<sizeof(LT);i++)
    if(overlap(O, LT[i]))
      remove(i, LT);
  putinLT(O,LT);//commit O
```

The `CLP(O, LT)` procedure scans the local locking table *LT* to find out all entries which are overlapping with the remote locking operation *O* and remove them from the *LT*, regardless of whether these entries belong to the local site or any other site. Finally, *O* is put in the *LT* and becomes committed.

3. When a new *Lock* operation is put into the *LT*, if the *LT* contains any overlapping *Lock* operations belonging to the same user, the new operation is merged with them.

6.4 Locking Region Adjustment

After executing an editing operation, whether a local or a remote one, the *Locking Region Adjustment (LRA)* procedure must be executed to adjust existing locking regions in the *LT* by taking into account of the impact of the newly executed editing operation.

```
LRA(O, LT) {
   for(int i=0; i<sizeof(LT); i++)
     IT(LT[i], O);
}
```

The `LRA(O, LT)` procedure transforms each *Lock* operation in the local locking table *LT* against the newly executed editing operation *O*. In this way, the locking regions in the *LT* can be kept consistent with the current document state.

7 Comparison to Related Work

Compared to locking schemes in traditional distributed computing and database systems, the locking scheme proposed in this paper for collaborative systems has the following important differences: First, the most fundamental difference is that locks in our system are used by intelligent human users rather than by preprogrammed processes, as in traditional distributed systems. Human users are capable of adjusting or redirecting their actions according to feedback from the underlying system. Second, after requesting a lock on a region, the human user is never blocked, and may go on updating the requested region or updating any other region, whereas the process issuing a lock request is blocked until the lock is granted or rejected in traditional locking systems. Third, a direct consequence of the second point is that there is no danger of deadlock in our system, whereas deadlock may occur in traditional locking systems. Fourth, contents in a locked region are visible (i.e., readable) to other users without a lock on the region in our system, but contents in a locked region are unreadable by other processes if the lock is a write-lock in traditional distributed systems.

Locking in other collaborative editing systems is all compulsory. The requirement for compulsory locking was largely due to a misconception that locking has a role to play for resolving all inconsistency problems in collaborative editors. Our research, however, has discovered that locking can help reduce conflict which may cause semantic inconsistency, but, at least in the text editing domain, locking has no role to play in resolving any syntactic inconsistency problem. Syntactic inconsistency problems have to be resolved by the operational transformation technique. Locking is made optional in our scheme because operational transformation has been used to ensure syntactic consistency (characterized by the CCI properties); locking is only occasionally needed to help avoid semantic conflicts.

Some existing locking schemes in collaborative editors are *optimistic* in the sense that a user is permitted to edit a region while waiting for the requested lock. Due to its nonblocking nature, optimistic locking is regarded to be well-suited to an collaborative environment where communication latency is high but conflict is rare [7], [8]. The optional locking scheme proposed in this paper is one step further toward a higher degree of optimism: Optimistic locking schemes are optimistic in the sense that they allow the users to edit a region without having obtained a lock (which is normally granted anyway), whereas, the optional locking scheme is more optimistic in the sense that it allows the users to edit a region even without requesting a lock (which is normally not needed anyway). If locking requests are normally granted (since conflicts are rare) and inconsistencies are repairable by human users (if conflicts do occur), why do we have to request a lock on a region in the first place?

The high responsiveness of our locking scheme is achieved by immediately granting the user a tentative lock and then applying either a distributed or a coordinator-based protocol to resolve conflict if multiple tentative locks happen to be concurrently placed on overlapping regions. In contrast to existing optimistic locking schemes [7], which undo concurrent updates if a (tentative) lock is eventually aborted, our responsive locking scheme preserves all concurrent updates during a transition locking period and leaves it to the users to decide what updates should be retained and what should be undone. We believe the strategy of preserving all users' work to facilitate a user-decided solution to conflict is more suitable to collaborative environments since conflicts among collaborative users are better understood and resolved by these users. Usage experiments are planned to evaluate this and other design decisions.

Fine-grain locking allows a high degree of concurrency in a collaborative editing session, but supporting fine-grain locking in the face of concurrent editing is nontrivial. To the best of our knowledge, the inconsistency problems associated with fine-grain locking operations have never been addressed by any existing locking scheme for collaborative systems. Our locking scheme is unique in applying and extending the operational transformation technique to support responsive and consistent fine-grain locking.

8 Conclusions

In this paper, we have contributed a novel optional and responsive fine-grain locking scheme for Internet-based real-time collaborative editing systems. We start from examining the role of locking in consistency maintenance to point out that (fine-grain) locking is unable to maintain syntactic consistency but is able to help maintain semantic consistency. Based on this observation and on the fact that operational transformation has been used for maintaining syntactic consistency, locking is made optional to save the users the burden of having to use locks while editing and the system the overhead of executing locking operations most of the time in a collaborative editing session. Then, a responsive locking scheme and protocols for conflict resolution have been proposed. Furthermore, a range of special technical issues related to achieving consistent fine-grain locking in collaborative text editors have been examined and resolved and the operational transformation technique has been extended to support consistent fine-grain locking. Algorithms for realizing the proposed schemes and protocols in the Internet environment have been also devised and discussed in detail. The similarities and differences of our locking scheme with existing locking schemes in traditional distributed computing and other collaborative editors are also discussed.

The optional and responsive fine-grain locking scheme has been implemented in the Web-based REDUCE (REal-time Distributed Unconstrained Cooperative Editing) system, which is publically demonstrated at http://reduce.qpsf.edu.au. By means of this prototype system, we are currently conducting usability study on the proposed locking scheme from end-users' perspective. In addition, we are working on the formal specification and verification of the schemes and algorithms presented in this paper.

Consistency maintenance is a fundamental issue in many areas of distributed computing. Research on Internet-based real-time collaborative computing systems has drawn inspirations from traditional distributed computing techniques (e.g., state-vector timestamping, locking, etc.) and has also invented nontraditional techniques (e.g., operational transformation and optional locking, etc.) to address special issues in distributed, interactive, and collaborative computing environments. The generalization and application of these nontraditional techniques to other areas of distributed computing and collaborative computing is an exciting direction for future exploration.

ACKNOWLEDGMENTS

The author wishes to thank Lingzhong Zhou and Dashan Zhou for their assistance in implementing the optional and responsive fine-grain locking scheme in the Internet-based REDUCE system. The work reported in this paper has been partially supported by an ARC (Australia Research Council) Large Grant (A00000711).

REFERENCES

[1] P. Bernstein, N. Goodman, and V. Hadzilacos, *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[2] R.M. Baecher et al, "The User-Centered Iterative Design of Collaborative Writing Software," *Proc. ACM INTERCHI Conf. Human Factors in Computing Systems,* pp. 399-405, 1993.

[3] K. Birman, A. Schiper, and P. Stephenson, "Lightweight Causal and Atomic Group Multicast," *ACM Trans. Computer Systems,* vol. 9, no. 3, pp. 272–314, Aug. 1991.

[4] J.D. Campbell, "Usability and Interference for Collaborative Diagram Development," *Proc. ACM Workshop Collaborative Editing Systems (CSCW '2000),* 2000.

[5] C.A. Ellis and S.J. Gibbs, "Concurrency Control in Groupware Systems," *Proc. ACM SIGMOD Conf. Management of Data,* pp. 399-407, 1989.

[6] C.A. Ellis, S.J. Gibbs, and G.L. Rein, "Groupware: Some Issues and Experiences," *Comm. ACM,* vol. 34, no. 1, pp. 39-58, Jan. 1991.

[7] S. Greenberg et al., "Issues and Experiences Designing and Implementing Two Group Drawing Tools," *Proc. Hawaii Int'l Conf. Systems Sciences,* pp. 138-150, 1992.

[8] S. Greenberg and D. Marwood, "Real Time Groupware as a Distributed System: Concurrency Control and Its Effect on the Interface," *Proc. ACM Conf. Computer Supported Cooperative Work,* pp. 207-217, Nov. 1994.

[9] A. Karsenty and M. Beaudouin-Lafon, "An Algorithm for Distributed Groupware Applications," *Proc. 13th Int'l Conf. Distributed Computing Systems,* pp. 195-202, May 1993.

[10] M. Knister and A. Prakash, "Issues in the Design of a Toolkit for Supporting Multiple Group Editors," *J. Usenix Assoc.,* vol. 6, no. 2, pp. 135-166, Spring 1993.

[11] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, July 1978.

[12] L.J. McGuffin and G.M. Olson, "ShrEdit: A Shared Electronic Workspace," CSMIL Technical Report #45, Univ. of Michigan, 1992.

[13] J. Munson and P. Dewan, "A Concurrency Control Framework for Collaborative Systems," *Proc. ACM Conf. Computer Supported Cooperative Work,* pp. 278-287, Nov. 1996.

[14] R.E. Newman-Wolfe et al., "MACE: A Fine Grained Concurrent Editor," *Proc. ACM COCS Conf. Organizational Computing Systems,* pp. 240-254,

[15] R.E. Newman-Wolfe et al., "Implicit Locking in the Ensemble Concurrent Object-Oriented Graphics Editor," *Proc. ACM Conf. Computer Supported Cooperative Work,* pp. 265-272, Nov. 1992.

[16] D. Nichols, P. Curtis, M. Dixon, and J. Lamping, "High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System," *Proc. ACM Conf. User Interface Systems and Tools,* pp. 111-120, Nov. 1995.

[17] A. Prakash and M. Knister, "A Framework for Undoing Actions in Collaborative Systems," *ACM Trans. Computer-Human Interaction,* vol. 4, no. 1, pp. 295-330, 1994.

[18] M. Raynal and M. Singhal, "Logical Time: Capturing Causality in Distributed Systems," *Computer,* pp. 49-56, Feb. 1996.

[19] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenbauser, "An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors," *Proc. ACM Conf. Computer Supported Cooperative Work,* pp 288-297, Nov. 1996.

[20] C. Sun and P. Maheshwari, "An Efficient Distributed Single-Phase Protocol for Total and Causal Ordering of Group Operations," *Proc. IEEE Third Int'l Conf. High Performance Computing,* pp. 295-300, Dec. 1996.

[21] 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.

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

[23] C. Sun and R. Sosie, "Optional Locking Integrated with Operational Transformation in Distributed Real-Time Group Editors," *Proc. 18th ACM Symp. Principles of Distributed Computing,* pp. 43-52, 1999.

[24] C. Sun, "Undo Any Operation at Any Time in Group Editors," *Proc. ACM Conf. Computer-Supported Cooperative Work,* pp. 191-200, 2000.

[25] R. Valdes, "Text Editors: Algorithms and Architectures, Not Much Theory, but a Lot of Practice," *Dr. Dobb's J.,* pp. 38-43, 1993.

[26] Y. Yang, C. Sun, Y. Zhang, and X. Jia, "Real-Time Cooperative Editing on the Internet," *IEEE Internet Computing,* pp. 18-25, May/June 2000.

Chengzheng Sun received a degree in wireless telecommunication technology from North-Eastern University, China, in 1976. He received an MPhil degree in computer engineering from East-China Institute of Computing Technology in 1982, the PhD degree in computer engineering from Changsha Institute of Technology, China, in 1987, and the PhD degree in computer science from the University of Amsterdam, The Netherlands, in 1992. He is a full professor (Chair of Internet Computing) in the School of Computing and Information Technology at Griffith University, Brisbane, Australia. His research interests and expertise include: Internet and Web computing technologies and applications, real-time groupware systems and CSCW (Computer-Supported Cooperative Work), distributed operating systems and computer networks, and parallel implementation of object-oriented and logic programming languages.