Achieving Convergence in Operational Transformation: Conditions, Mechanisms and Systems

Yi Xu, Chengzheng Sun, Mo Li
School of Computer Engineering, Nanyang Technological University, Singapore

ABSTRACT
In this paper, we present a comprehensive and in-depth study on convergence preservation and avoidance in Operational Transformation (OT) systems. In this study, we discovered basic conditions, transformation patterns, and mechanisms for avoiding Convergence Property 2 (CP2), and established CP2-avoidance correctness of seven major OT systems. Furthermore, we proposed improvements to existing systems and designed a new OT system capable of avoiding CP2 with a unique combination of novel features. These results contribute significantly to the advancement of OT and collaboration-enabling technology.

Author Keywords
Operational Transformation; consistency maintenance; convergence; real-time collaborative applications.

ACM Classification Keywords
H.5.3 [**Information Systems**]: Group and Organization Interfaces – Synchronous Interaction, Theory and Model.

INTRODUCTION

Consistency maintenance is a main challenge in building real-time collaborative applications over high latency communication environments like the Internet [4, 17]. Due to its non-blocking, fine-grained concurrency, and unconstrained interaction properties, Operational Transformation (OT) is particularly suitable for consistency maintenance in such environments, and has been increasingly adopted in industrial applications, e.g. Google Wave/Docs[1], IBM OpenCoWeb[2], and Codoxword[3].

In OT-supported real-time collaborative editing applications, multiple users may freely and simultaneously generate editing operations, which may be transformed and executed in different orders at different collaborating sites [1, 2, 3, 18, 24]. Document *convergence* is a key consistency requirement: the final documents at all sites must be *identical* after executing the same group of operations [18]. Past research has discovered important transformation properties for achieving convergence. In the first OT system [3], a transformation property, *Convergence Property 1* (*CP1*)[4], was identified, which ensures the same document is produced by executing two concurrent operations in different orders. Follow-up research found CP1 was insufficient to ensure convergence under certain circumstances, and added another transformation property, *Convergence Property 2* (*CP2*)[5], which ensures the same operation is produced in transforming one operation against two concurrent operations in different orders (see the next section for precise definitions of CP1 and CP2).

There exist two general approaches to achieving convergence in OT: one is to design application-specific transformation functions capable of preserving CP1/CP2; the other is to design generic control algorithms capable of avoiding CP1/CP2. Past research has found that it is relatively easy to preserve CP1 by transformation functions and to avoid CP2 by control algorithms. Multiple CP1-preserving transformation functions for a variety of data and operation models have been designed [1, 11, 14, 20, 24]; and numerous control algorithms capable of avoiding CP2 have been invented [8, 10, 15, 16, 18, 21, 23]. An OT control algorithm is said to be able to avoid CP2 if it works with transformation functions capable of only preserving CP1 for achieving convergence. There were some attempts to design transformation functions for preserving CP2 in character-wise text editing systems [6, 7, 12], but avoiding CP2 by control algorithms has been favored due to its advantage of being generic and applicable to different data and operation models for supporting a range of applications. Recent OT systems for advanced collaborative applications, e.g. 2D spreadsheets [22] and 3D digital media design systems [1], have been designed with a combination of transformation functions[6] for preserving CP1 and control algorithms for avoiding CP2 [21].

Despite the wide adoption of the CP2-avoidance strategy in OT systems, the basic conditions and mechanisms for CP2-avoidance have not been well-understood. Most OT systems that claimed CP2-avoidance had justified their claims by showing the absence of transforming an operation against the same pair of operations in different orders, which is intuitive-

[1] http://docs.google.com
[2] https://github.com/opencoweb/coweb#readme
[3] http://www.codoxware.com

CSCW'14, February 15 – 19, 2014, Baltimore, MD, USA.

http://dx.doi.org/10.1145/2531602.2531629

[4] CP1 was named as *Transformation Property* (*TP*) in [3], and *Transformation Property 1* (*TP1*) in [14].
[5] CP2 was named as *Transformation Property 2* (*TP2*) in [14], which is the same as *Transpose Property 5* proposed in [11].
[6] Transformation functions must preserve, in addition to generic CP1, application-specific *combined effects* for concurrent operations [1, 20, 22] or *operation intentions* [18].

ly derived from CP2 (see details in the next section) [8, 15, 23]. However, more recent research has found the mere absence of transforming the same pair of operations in different orders is *insufficient* to avoid CP2 (see the next section for a counter-example), and a correct condition of avoiding CP2 is: *to avoid transforming two operations in different contexts* [21]. The discovery of this condition has advanced OT knowledge, but also cast doubt on the CP2-avoidance claims of prior OT systems: are they really able to avoid CP2? After all, they were either not verified or verified by an insufficient condition. Resolving this doubt is important as it is related to the correctness of a large number of major OT systems [8, 10, 15, 16, 18, 21, 23, 24], some of which are being used in industrial applications with millions of users, e.g. the Jupiter OT [10] is the basis for Google OT [24, 16], which is used for Google Docs; and COT [21] is used in Codoxword and IBM OpenCoWeb. As OT is increasingly applied to a wider range of real-world applications and used by more and more people, ensuring and verifying correctness of its core algorithms become more and more important.

To address these open issues, we set out to study seven major OT systems that claimed or was recognized for being capable of avoiding CP2, including Jupiter[7][10], NICE[15], Google OT [24,16], SOCT3 and SOCT4 [23], TIBOT [8], and COT [21], with the following objectives: (1) to validate whether they indeed meet the correct CP2-avoidance condition; and (2) to find out how and why they did it (if they can). From this study, we have not only achieved the original objectives, but also made important discoveries in CP2-avoidance conditions, and basic transformation patterns and mechanisms that can be used to avoid CP2 in different OT systems. Furthermore, we proposed improvements to examined systems and designed a new OT system capable of avoiding CP2 with a unique combination of novel features. We report these research findings in this paper.

The rest of this paper is organized as follows. First, we introduce background knowledge related to this work. Then, we discuss CP2-avoidance conditions. Next, we present basic CP2-avoidance transformation patterns and mechanisms. Based on these patterns and mechanisms, we present our CP2-avoidance validation of seven systems. Furthermore, we propose improvements to Jupiter and Google OT, present the design of a new OT system, and discuss ideas of future work inspired by this study. Finally, we summarize the main contributions of this work.

BACKGROUND KNOWLEDGE

Causal and Total Ordering Relations

Following Lamport [5], the causal ordering relation of operations is defined as follows [3].

Definition 1. Causal Relation "→". *Given two operations O_a and O_b, generated at site i and j, respectively, O_a is causally before O_b, denoted by $O_a \rightarrow O_b$, if and only if: (1) i=j and the generation of O_a happened before the generation of O_b; (2) i≠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. Concurrent Relation "||". *Given two operations O_a and O_b, O_a and O_b are concurrent, denoted by $O_a \| O_b$, if and only if neither $O_a \rightarrow O_b$ nor $O_b \rightarrow O_a$.*

In real-time collaborative editing systems, operations with causal relationships are executed in their causal orders; but concurrent operations may be executed in any orders [3, 17]. In OT systems, concurrent operations are transformed to achieve consistency, but transformation orders are also governed by causal relations among operations.

Definition 3. Total Ordering Relation "⇒". *A relation "⇒" is a total ordering relation among operations in a session if the following statements hold for any operations O_a, O_b, and O_c, where $O_a \neq O_b \neq O_c$: (1) if $O_a \Rightarrow O_b$ and $O_b \Rightarrow O_c$, then $O_a \Rightarrow O_c$ (transitivity); and (2) $O_a \Rightarrow O_b$ or $O_b \Rightarrow O_a$ (totality).*

In contrast to the causal relation, which defines a partial ordering relation among operations, a total ordering relation is a complete ordering relation among operations. The "totality" condition in the total ordering relation means that all operations are ordered under "⇒". In this work, we are particularly interested in a special class of total ordering relations that are consistent with the causal ordering relation among the same group of operations, i.e. for any O_a and O_b, if $O_a \rightarrow O_b$, then $O_a \Rightarrow O_b$. In the rest of this paper, we assume "⇒" is always consistent with "→". As will become clear later, such total ordering relations play a key role in defining CP2-avoidance conditions.

Operation Context and Context-based Conditions

In OT systems, every operation O is associated with a *context*, which represents the document state on which O is defined [17, 18, 21]. For the purpose of OT design, the context of O needs not be represented by a real document state, but can be represented by a *set* of operations (in their original forms) that have been executed to create the state on which O is defined [21].

Definition 4. Document State Representation. *A document state, denoted as S, can be represented as follows:(1) the initial document state is represented as an empty set $S = \{\}$; (2) after executing an operation O on the document, the new state is represented by: $S = S \cup \{org(O)\}$, where $org(O)$ represents the original form of O; if O is an original operation (generated by a user), $org(O) = O$.*

Definition 5. Operation Context Representation. *The context of an operation O, denoted as $C(O)$, is represented as follows: (1) for an original operation O, $C(O) = S$, where S is the document state from which O is generated; (2) for a transformed operation O', $C(O') = C(O) \cup \{org(Ox)\}$, where $O' = T(O, Ox)$, and T is a function that transforms O into O' according to the impact of Ox.*

The significance of operation context is that it provides the ground for interpreting the effect of an operation and for

[7]When Jupiter OT was published in 1994 [10], CP2 had not yet be identified as an OT convergence requirement. So Jupiter OT was not designed to purposely avoid CP2, but to rely solely on CP1 to achieve convergence.

reasoning about the relations among operations, which are essential for ensuring correct operation execution and transformation [17, 18, 21]. Among others, two context-based conditions are directly relevant to this work:

1. *Context-equivalence condition for execution*: given an operation O and a document state S, O can be correctly executed on S only if $C(O) = S$.
2. *Context-equivalence condition for transformation*: given two operations O_1 and O_2, they can be correctly transformed with each other, by invoking $T(O_1, O_2)$ or $T(O_2, O_1)$, only if $C(O_1) = C(O_2)$.

In theory, any operation (either original or transformed) has a context, but operations included in a context are all in original forms. Causal and total ordering relations are defined only among original operations [21]. For notation conciseness, however, we assume an automatic operation form conversion, from "O" to "$org(O)$", whenever "→", "||", "⇒" and other context set operators (e.g. "∪") are used, e.g. "$O_a \Rightarrow O_b$" means "$org(O_a) \Rightarrow org(O_b)$". When we say an operation O is in an operation context, we mean $org(O)$ is in a context. We use the notation $O_a\{b\}$ to denote that O_b is in the context of O_a, which can be used to represent the result of transforming O_a against O_b, i.e. $O_a\{b\} = T(O_a, O_b)$. If O is generated from an initial document state, no operation is included in its context, so it can be expressed as $O\{\}$ or simply O.

Convergence Properties

Convergence Property 1 (CP1): Given O_a and O_b defined on the state S, if $O_a\{b\} = T(O_a, O_b)$, and $O_b\{a\} = T(O_b, O_a)$, the following holds:

$$S \circ O_a \circ O_b\{a\} = S \circ O_b \circ O_a\{b\},$$

which means the same document is produced by applying O_a and $O_b\{a\}$ in sequence on S, and applying O_b and $O_a\{b\}$ in sequence on S, respectively.

Convergence Property 2 (CP2): Given O_a, O_b and O_c defined on the same state, if $O_c\{b\} = T(O_c, O_b)$ and $O_b\{c\} = T(O_b, O_c)$, the following holds:

$$T(T(O_a, O_b), O_c\{b\}) = T(T(O_a, O_c), O_b\{c\}),$$

which means the same operation is produced by transforming O_a against O_b and then $O_c\{b\}$, and transforming O_a against O_c and then $O_b\{c\}$, respectively.

CP2-AVOIDANCE CONDITIONS

An Insufficient CP2-Avoidance Condition

CP2 requires the same operation is produced in transforming one operation against the same pair of operations in different orders, so it is intuitive to derive: *CP2 can be avoided if an OT system can avoid transforming an operation against the same pair of operations in different orders.* This condition has been explicitly or implicitly used to validate CP2-avoidance in several OT systems [8, 15, 23], but it is insufficient to achieve convergence.

A counter-example to this condition is given in Figure 1. In this scenario, there are three operations: $O_1 = I(2, x)$ (to insert "x" at position *2*), $O_2 = D(1, b)$ (to delete "b" at position *1*), and $O_3 = I(1, y)$ (to insert "y" at position *1*), concurrently generated from the same initial document state "*abc*". All possible execution and transformation paths of these operations are illustrated in a Context-based Operation Transformation Space (COTS)[8] in Figure 1.

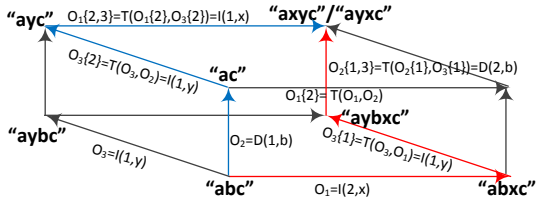

Figure 1: A counter-example to the insufficient CP2-avoidance condition (assuming *priority*(O_1) > *priority*(O_3)).

Suppose an OT system forces operation execution and transformation along only two paths in Figure 1:

1. In one path, O_2, $O_3\{2\}$ and $O_1\{2, 3\}$ are executed in sequence, where:
   (1) $O_3\{2\} = T(O_3, O_2) = I(1, y)$;
   (2) $O_1\{2, 3\} = T(O_1\{2\}, O_3\{2\}) = I(1, x)$;
   which results in a final state "axyc";
2. In another path, O_1, $O_3\{1\}$ and $O_2\{1,3\}$ are executed in sequence, where:
   (1) $O_3\{1\} = T(O_3, O_1) = I(1, y)$;
   (2) $O_2\{1, 3\} = T(O_2\{1\}, O_3\{1\}) = D(2, b)$;
   which results in a final state "ayxc".

Clearly, no operation has ever been transformed against two other operations in different orders along these two paths, but the final states produced by the two paths are not identical – convergence is not ensured, which means the condition is insufficient to achieve convergence.

A General CP2-Avoidance Condition

A correct and general CP2-avoidance condition was discovered in [21], which can be stated as: *CP2 can be avoided if an OT system can avoid transforming the same pair of operations in different contexts*.

It can be seen that transformations along the two divergent paths in Figure 1 violate this condition:

1. O_2 and O_3 are transformed under different contexts: $T(O_3, O_2)$ in 1-(1) vs $T(O_2\{1\}, O_3\{1\})$ in 2-(2).
2. O_1 and O_3 are also transformed under different contexts: $T(O_1\{2\}, O_3\{2\})$ in 1-(2) vs $T(O_3, O_1)$ in 2-(1).

The condition in [21] requires the contexts of any two operations are always the *same* whenever they are transformed with each other, but it does not specify what concrete conditions the contexts of two operations should meet when they are transformed. This CP2-avoidance condition is general but unsuitable for direct use in validating specific OT systems. In this work, we discovered a special CP2-avoidance condition that satisfies the general condition and also defines concrete

[8] In a COTS, a vertex represents a document state, and an arrow represents an operation: the starting vertex of the arrow represents the definition state of the operation, and the ending vertex represents the resulting state of the operation. A correct COTS is one that ensures there is only one label for each vertex or arrow.

context conditions between two operations for avoiding CP2; and we use this new condition to validate seven OT systems.

A Special CP2-Avoidance Condition

Definition 6. Totally Ordered and Equivalent Context Relation "≌". *Given a "⇒" among all operations in a session, two operations O_a and O_b have a Totally Ordered and Equivalent Context Relation, denoted by $C(O_a)$ ≌ $C(O_b)$, when the following conditions are met: (1) $O_a \Rightarrow O_b$; (2) $C(O_a) = C(O_b)$; and (3) O_x is included in $C(O_a)$ and $C(O_b)$ if and only if:*

1. *$O_x \Rightarrow O_a \Rightarrow O_b$; or*
2. *$O_a \Rightarrow O_x \Rightarrow O_b$ and $O_x \rightarrow O_b$.*

The "≌" relation requires the contexts of O_a and O_b include all and only those operations that are totally before both O_a and O_b, and those that are totally between O_a and O_b and causally before O_b. The following theorem establishes the uniqueness of the context under this relation.

Theorem 1. *For any two operations O_a and O_b in a session, operations included in their contexts are uniquely defined if $C(O_a)$ ≌ $C(O_b)$.*

Proof: In a given session, the causal ordering relation "→" between any pair of operations must be fixed; and, under a given total ordering relation "⇒", the total ordering among all operations must be unique by Definition 3. The theorem follows directly from the uniqueness of the "⇒" and "→" relations among operations in the same session and the definition of "≌".

The general CP2-avoidance condition can be restated as follows: *CP2 can be avoided if an OT system can ensure the same pair of operations are transformed always on the same context*. This restatement highlights the *context uniqueness* essence of CP2-avoidance. The following theorem establishes a special CP2-avoidance condition.

Theorem 2. *CP2 can be avoided if an OT system can ensure a pair of operations O_a and O_b are transformed only if $C(O_a)$ ≌ $C(O_b)$.*

Proof: It is derived directly from the general CP2-avoidance condition and the uniqueness of the context of O_a and O_b under $C(O_a)$≌$C(O_b)$ (Theorem 1).

We will show this special condition is general enough to cover all existing OT systems capable of avoiding CP2.

CP2 AVOIDANCE MECHANISMS AND PATTERNS

In this section, we discuss the basic mechanisms and transformation patterns that are required to meet the special CP2-avoidance condition.

Total Ordering Schemes

A total ordering relation "⇒" among operations in a session is at the core of the special CP2-avoidance condition, so a total ordering scheme is a basic mechanism for achieving CP2-avoidance in OT systems.

For a group of n operations in a session, there may be a maximum of $n!$ ways of totally ordering them. The special CP2-avoidance condition only requires the same "⇒" is used to order all operations in the same session, without imposing constraints on specific total ordering relations or ways of creating such total ordering relations. We describe three types of total ordering schemes below.

An OT system may use an *implicit* total ordering scheme based on a central server for broadcasting operations among collaborating sites: each site connects and sends local operations to the server via a FIFO (First In First Out) communication channel (e.g. a TCP connection); and the central server serializes operations and broadcasts them among all sites. The operation *serialization* order at the server *implicitly* forms a total order among all operations. This kind of total ordering scheme can be found in Jupiter [10], Google OT for Wave/Docs [16, 24], NICE [15], and Codoxword [21].

An OT system may also use an *explicit* total ordering scheme based on a special sequencer, which does not broadcast operations but only generates continuous sequence numbers for ordering operations. After generating a local operation, a collaborating site requests the central sequencer for a sequence number to timestamp this operation. In this way, all operations are totally ordered by sequence numbers. This total ordering scheme is used in SOCT3 and SOCT4 [23].

A total ordering scheme can be based on a distributed timestamping scheme without involving any central server/sequencer: collaborating sites use special rules to broadcast and timestamp operations so that a total ordering relation among all operations can be unambiguously derived. One example of such distributed schemes is the Time-Interval-Based OT system TIBOT [8].

One way or another, all existing OT systems that claimed or were recognized for CP2-avoidance have a total ordering scheme. However, the formation and use of such total ordering schemes may be implicit in these systems.

Operation Sequence and Transformation Patterns

We have identified several basic operation sequences and transformation patterns that can be used to ensure $C(O_a)$≌ $C(O_b)$ in transformations, which are described below.

Definition 7. Totally-Ordered and Contextualized Operation Sequence (TOCOS). *Given a list of operations $L = [O_1, O_2, ..., O_n]$ and a "⇒" among all operations in a session, L is a TOCOS if it is an empty list or a list with one operation, or the following holds (when $n \geq 2$): (1) $O_i \Rightarrow O_j$, for $1 \leq i < j \leq n$; and (2) $C(O_{i+1}) = C(O_i) \cup \{O_i\}$, for $1 \leq i < n$.*

All operations (if any) in a TOCOS are required to be totally ordered and contextualized, i.e. the context of an operation O_{i+1} must include O_i and all operations in the context of O_i. Contextualization can be achieved at the time of operation generation if $O_i \rightarrow O_{i+1}$, or by transformation if they are concurrent, i.e. $O_i \| O_{i+1}$.

Based on TOCOS, we define two basic operation sequence patterns related to avoiding CP2.

Definition 8. ***Operation Sequence Pattern 1 (OSP1).*** *Given a TOCOS $L=[O_1, O_2, ..., O_n]$ and an operation O_x that is not in L, L is an OSP1 with respect to O_x if L is an empty list or the following conditions hold: (1) $C(O_1) \trianglerighteq C(O_x)$; (2) $O_n \Rightarrow O_x$; and (3) L includes every operation O_y in a session, such that $O_y \parallel O_x$ and $O_1 \Rightarrow O_y \Rightarrow O_n$ (when n≥2).*

The first operation (if any) in L must have the "$\trianglerighteq$" relation with O_x; all operations in L must be totally ordered before O_x; all operations that are concurrent with O_x and totally ordered between O_1 and O_n must be included in L.

Definition 9. ***Operation Sequence Pattern 2 (OSP2).*** *Given a TOCOS $L = [O_1, O_2, ..., O_n]$ and an operation O_x that is not in L, L is an OSP2 with respect to O_x if L is an empty list or the following conditions hold:(1) $C(O_x) \trianglerighteq C(O_1)$; (2) $O_i \rightarrow O_j$, for $1 \le i < j \le n$ (when $n \ge 2$); and (3) L includes every operation O_y in the session, such that $O_1 \rightarrow O_y \rightarrow O_n$.*

O_x must have the "$\trianglerighteq$" relation with the first operation (if any) in L, which implies O_x is totally ordered before all operations in L; all operations in L must be causally related; and all operations that are causally ordered between O_1 and O_n must be included in L.

The following *LT* function transforms an operation O against a sequence of operations in L to produce $O\{L\}$.

List Transformation: $LT(O, L) = O\{L\}$
1. $O\{L\} := O$;
2. **for** ($i = 1$; $i \le |L|$; i++) {
   $O\{L\} := T(O\{L\}, L[i])$;}

The *SLT* function symmetrically transforms an operation O with operations in L to produces $O\{L\}$ and $L\{O\}$.

Symmetric List Transformation: $SLT(O, L)=(O\{L\}, L\{O\})$
1. $O\{L\} := O$;
2. **for** ($i = 1$;$i \le |L|$; i++) {
   $L\{O\}[i] := T(L[i], O\{L\})$;
   $O\{L\} := T(O\{L\}, L[i])$; }

$LT(O_x, L)$ and $SLT(O_x, L)$ define two basic CP2-avoidance transformation patterns under the condition that L is an *OSP1* or *OSP2* with respect to O_x, which is established by the following theorem.

Theorem 3. *If L is an OSP1 or OSP2 with respect to O_x, CP2 is avoided in all transformations in $LT(O_x, L)$ and $SLT(O_x, L)$.*

Proof: Consider the transformation pattern: $LT(O_x, L)$ where L is an *OSP1* with respect to O_x. From Definition 8, we know $C(L[1]) \trianglerighteq C(O_x)$. Therefore, CP2 can be avoided in $T(O_x, L[1])$ by Theorem 2. After this transformation, O_x becomes $O_x\{L[1]\}$, which implies $C(L[2]) \trianglerighteq C(O_x\{L[1]\})$ by Definition 6 and Definition 8, hence CP2 can be avoided in $T(O_x\{L[1]\}, L[2])$. By a deductive argument, CP2 can be avoided in every transformation in $LT(O_x, L)$. Following the same reasoning, CP2 can be avoided in all transformation patterns stated in the theorem.

VALIDATING OT SYSTEMS FOR CP2 AVOIDANCE

One major challenge in this work is the diversity of the systems to be examined: they were designed for different purposes and had major differences in: (1) system structures, e.g. some use a central server and some do not; (2) operation propagation protocols, e.g. some delay propagating operations until certain conditions are met, and some do not; and (3) mechanisms and algorithms, e.g. some use state/context-vectors, and some do not; some are based on two-dimensional (2D) state-spaces, and some use one-dimensional (1D) buffers. These systems were designed by different research groups and described in different styles and terminologies, which present challenges to figure out their real similarities and differences.

Our approach to addressing these challenges is to devise a general framework encapsulating the basic mechanisms and transformation patterns that may occur at three critical locations: (1) local sites, where operations are generated; (2) the server site (optional), where operations are broadcast to other sites; and (3) remote sites, where operations generated by other sites are replayed. Then, we dissect the seven OT systems one-by-one to extract relevant schemes and algorithms, and describe them uniformly in terms of the basic mechanisms and patterns under the framework. Finally, we validate CP2-avoidance by matching their transformation patterns with CP2-avoiding transformation patterns established in the prior section.

In addition, we use a running example to illustrate how the seven systems work in concrete scenarios, in which there are three collaborating sites (plus an optional server) and four operations with causal relations: $(O_1 \rightarrow O_4) \parallel (O_2 \rightarrow O_3)$, and total ordering relations: $O_1 \Rightarrow O_2 \Rightarrow O_3 \Rightarrow O_4$.

Jupiter

In the Jupiter system [10], collaborating clients are connected to a central server via TCP connections in a star-like topology. A client can communicate directly with the server only; the server sequentially processes and broadcasts operations among all clients.

Local processing

When O_x is generated by a client, it is executed immediately at the local site and propagated to the server as-is. Local operations are propagated sequentially and saved along the local dimension of a 2D state-space.

Server processing

The server maintains multiple 2D state-spaces[9], one for every client. A state-space consists of a *local* dimension for operations from the corresponding client, and a *global* dimension for operations from all other sites. The server handles O_x as follows:

1. locates the space for the client from which O_x comes;
2. searches this space to find the state that matches the context of O_x, and saves O_x at this matching state along the local dimension;

[9]The multiple 2D state-spaces maintenance was not reported in the Jupiter paper [10]. The description here is based on our understanding and derivation from other sources [9, 16, 24].

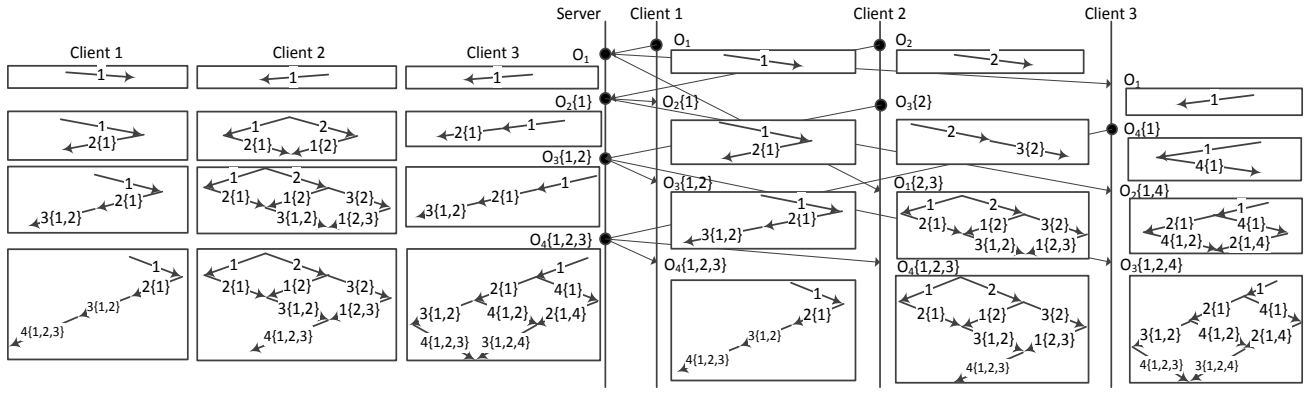

Figure 2: Jupiter: the server maintains three 2D state-spaces; each client maintains a single state-space; in a 2D state-space, an operation $O_n\{m\}$ is represented by arrow labeled by $n\{m\}$ (for conciseness), the local dimension is represented by south-east arrows, and the global dimension represented by south-west arrows.

3. transforms O_x symmetrically with a sequence of operations, denoted as L_1, consisting of operations along the global dimension from the matching state to the final state of the space; this transformation can be expressed as: $SLT(O_x, L_1) = (O_x\{L_1\}, L_1\{O_x\})$[10];
4. propagates $O_x\{L_1\}$ to other clients;
5. saves $O_x\{L_1\}$ at the end of the global dimension of every other state-space maintained by the server.

Remote processing

Each client maintains a state-space similar to that at the server. Jupiter handles $O_x\{L_1\}$ at a remote site as follows:

1. searches the space to find a matching state for $O_x\{L_1\}$, and saves $O_x\{L_1\}$ at this matching state along the global dimension;
2. transforms $O_x\{L_1\}$ symmetrically with a sequence of operations, denoted as L_2, consisting of operations along the local dimension from the matching state to the final state of the space; this transformation can be expressed as $SLT(O_x\{L_1\}, L_2)=(O_x\{L_1, L_2\}, L_2\{O_x\})$;
3. executes $O_x\{L_1, L_2\}$.

CP2-avoidance validation

It can be shown that L_1 is an OSP1 with respect to O_x for $SLT(O_x, L_1)$, and L_2 is an OSP2 with respect to $O_x\{L_1\}$ for $SLT(O_x\{L_1\}, L_2)$ because:

1. all operations in L_1 and L_2 are:
   (1) totally ordered by the server serialization;
   (2) contextualized by transformations in *SLT;*
   (3) concurrent with O_x as they are generated by other clients before O_x reaches these clients, and O_x is generated before they reach O_x's local site;
2. L_1 contains all operations that are totally before O_x, and $L_1[1] \preceq O_x$;
3. L_2 contains operations that are totally after O_x and causally related with each other, and $O_x\{L_1\} \preceq L_2[1]$.

By Theorem 3, Jupiter CP2-avoidance is confirmed.

[10] For conciseness, details of using intermediate transformation results created during *SLT/LT* to update 2D state-spaces and other buffers for all systems are omitted.

An example

Consider operation $O_3\{2\}$ at Client *2* in Figure 2. Upon generation, it is executed immediately, appended at the end of the local dimension of a 2D state-space, and propagated to the server. At the server, the state-space for Client *2* is firstly located and searched to find the matching state at the end of O_2. Then, $O_3\{2\}$ is saved after O_2 along the local dimension, and symmetrically transformed with $O_1\{2\}$ along the global dimension, which produces two transformed operations: $O_3\{1,2\}$ and $O_1\{2,3\}$, which are placed in corresponding dimensions of the state-space. $O_3\{1,2\}$ is propagated to Clients *1* and *3,* and appended to the ends of global dimensions of their state-spaces as well. When $O_3\{1,2\}$ arrives at Client *3,* it is saved at the matching state at the end of $O_2\{1\}$ along the global dimension, then symmetrically transformed with $O_4\{1,2\}$ along the local dimension, which produces two transformed operations: $O_3\{1,2,4\}$ and $O_4\{1,2,3\}$ in corresponding dimensions of the state-space. Finally, $O_3\{1,2,4\}$ is executed at Client *3*.

NICE

The NICE system [15] is also based on a central server for serializing and broadcasting operations. NICE supports OT-based flexible notification for both real-time and non-real-time collaboration. We extract and describe only the relevant consistency maintenance aspects below.

Local processing

NICE handles local operations in the same way as Jupiter, except using a 1D buffer to save local operations.

Server processing

The server maintains multiple 1D buffers, one for every client to maintain operations that are from other clients and have been broadcast. The server handles O_x as follows:

1. locates the buffer for the client from which O_x comes;
2. searches the buffer to find the matching state for O_x;
3. transforms O_x symmetrically with a sequence of operations, denoted as L_1, starting from matching state to the end of the buffer; this transformation can be expressed as: $SLT(O_x, L_1) = (O_x\{L_1\}, L_1\{O_x\})$;
4. propagates $O_x\{L_1\}$ to other clients;
5. saves $O_x\{L_1\}$ at the end of every other 1D buffer.

Remote processing

NICE handles $O_x\{L_1\}$ at a remote site as follows:

1. searches the buffer for the state that matches the context of $O_x\{L_1\}$ (this state may not exist);
2. transforms $O_x\{L_1\}$ symmetrically with a sequence of operations, denoted as L_2, consisting of operations from the matching state (if exists) to the end of the buffer; this transformation can be expressed as $SLT(O_x\{L_1\}, L_2) = (O_x\{L_1, L_2\}, L_2\{O_x\})$;
3. executes $O_x\{L_1, L_2\}$ (no need to save it in the buffer).

CP2-avoidance validation

It can be shown that L_1 is an OSP1 with respect to O_x in $SLT(O_x, L_1)$; and L_2 is an OSP2 with respect to $O_x\{L_1\}$ in $SLT(O_x\{L_1\}, L_2)$ by the same reasoning as that for Jupiter. By Theorem 3, NICE CP2-avoidance is confirmed.

An example

Consider operation $O_3\{2\}$ at Client *2* in Figure 3. Upon generation, it is executed immediately at Client *2*, appended at the end of the local buffer, and propagated to the server. At the server, the buffer for Client *2* is located and searched to find the matching state at beginning of the buffer. Then, $O_3\{2\}$ is symmetrically transformed with $O_1\{2\}$, which produces two transformed operations: (1) $O_3\{1,2\}$ is propagated to Clients *1* and *3,* and appended to the ends of their buffers as well; (2) $O_1\{2,3\}$ is used to replace $O_1\{2\}$ in the buffer for Client *2*. When $O_3\{1,2\}$ arrives at Client *3,* it is symmetrically transformed with $O_4\{1,2\}$ since they share the same state, which produces two transformed operations: $O_3\{1,2,4\}$ is executed; and $O_4\{1,2,3\}$ is used to replace $O_4\{1,2\}$ in the buffer.

Google OT for Google Wave and Docs

Google OT for Google Wave and Docs [9, 16, 24] is adapted from Jupiter, with the same star-like architecture and a central server for serializing and broadcasting operations. Google OT adds a *stop-and-wait* propagation protocol between each client and the server. As a result, a single 1D buffer at the server is adequate to maintain all possible states for transformation, which eliminates multiple 2D state-spaces in Jupiter.

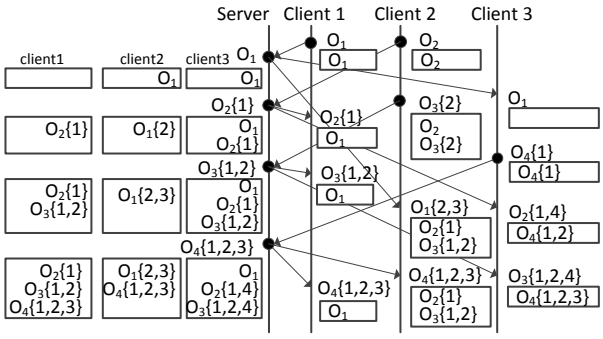

Figure 3: NICE: the server maintains three 1D buffers; each client maintains one buffer.

Local processing

When O_x is generated by a client, it is executed immediately and saved in the local dimension of a 2D state-space. However, O_x is not propagated to the server until the prior local operation O_{x-1} (if any) has been acknowledged by the server. Consequently, when O_x is propagated, it must have been symmetrically transformed with a sequence of remote operations (if any), denoted as L_1, that are propagated from the server (see the follow-up description for remote processing) and totally before O_{x-1} (hence before O_x as well). This transformation can be expressed as: $SLT(O_x, L_1) = (O_x\{L_1\}, L_1\{O_x\})$; the transformed operation $O_x\{L_1\}$ is propagated to the server.

Server processing

The server maintains a single 1D buffer for keeping track operations from all clients in a totally ordered and contextualized fashion. The server handles $O_x\{L_1\}$ as follows:

1. searches the 1D buffer to find the state that matches the context of $O_x\{L_1\}$;
2. transforms $O_x\{L_1\}$ against a sequence of operations, denoted as L_2, from the matching state to the end of the buffer; this transformation can be expressed as $LT(O_x\{L_1\}, L_2) = O_x\{L_1, L_2\}$;
3. appends $O_x\{L_1, L_2\}$ at the end of the buffer (other operations in the buffer remain unchanged); and
4. propagates $O_x\{L_1, L_2\}$ to all sites, including the originating site of $O_x\{L_1\}$ as an acknowledgement.

Remote processing

At each client site, there is a 2D state-space similar to Jupiter. The handling of $O_x\{L_1,L_2\}$ at a remote site is also similar to handling $O_x\{L_1\}$ in Jupiter: $O_x\{L_1,L_2\}$ will be symmetrically transformed with a sequence of operations, denoted as L_3, consisting of operations along the local dimension from the state that matches the context of $O_x\{L_1,L_2\}$ to the final state of the state-space. This transformation can be expressed as $SLT(O_x\{L_1,L_2\},L_3) = (O_x\{L_1, L_2,L_3\}, L_3\{O_x\})$. Finally, $O_x\{L_1,L_2,L_3\}$ is executed.

CP2-avoidance validation

It can be shown that L_1 is an OSP1 with respect to O_x in $SLT(O_x, L_1)$, L_2 is an OSP1 with respect to $O_x\{L_1\}$ in $LT(O_x\{L_1\}, L_2)$, and L_3 is an OSP2 with respect to $O_x\{L_1, L_2\}$ in $SLT(O_x\{L_1,L_2\}, L_3)$ by following the same reasoning as for Jupiter. By Theorem 3, Google OT CP2-avoidance is confirmed.

An example

Consider operation $O_3\{2\}$ at Client *2* in Figure 4. Upon generation, it is executed immediately and appended at the end of the local dimension of a 2D state-space. It is not propagated to the server until it has been transformed with O_1 and Client *2* has received the acknowledgement of $O_2\{1\}$, which is the prior operation causally before $O_3\{2\}$. The transformed operation $O_3\{1,2\}$ is propagated to the server. At the server, the matching state is located at the end of $O_2\{1\}$ in the buffer and there is no other operation shares this state, so $O_3\{1,2\}$ is appended to the end of the buffer and then propagated to all

clients. When $O_3\{1,2\}$ arrives at Client *3,* it is handled in exactly the same way as in Jupiter.

Consider another operation $O_4\{1\}$ at Client *3* in Figure 4. Upon generation, it is executed immediately and saved. It is immediately propagated to the server since it is the first local operation at Client *3*. At the server, the matching state is at the end of O_1 in the buffer, so $O_4\{1\}$ is transformed with $O_2\{1\}$ and $O_3\{1,2\}$ in sequence to produce $O_4\{1,2,3\}$, which is appended at the end of the buffer and propagated to all clients. When $O_4\{1,2,3\}$ arrives at Client *1,* it is executed without transformation and appended at the end of the state-space along the global dimension.

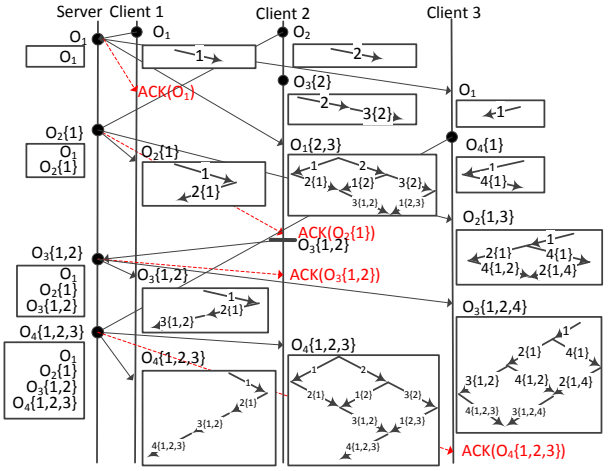

Figure 4: Google OT: the server maintains a single linear buffer; each client maintains one 2D state-space; local operations are acknowledged by the server.

SOCT3

Different from all prior systems, SOCT3 uses a central sequencer, which issues continuous sequence numbers to totally order operations, but is not involved in transforming or broadcasting operations. In addition, SOCT3 also uses state-vectors (SV) to capture concurrency relations among operations.

Local processing

When O_x is generated at a site, it is executed immediately, and timestamped by a sequence number acquired from the sequencer and a state-vector. Local operations are propagated as-is in sequence and saved in a history buffer (*HB*).

Remote processing

SOCT3 handles O_x at a remote site as follows:

1. waits until all operations with smaller sequence numbers have been executed at this site;
2. copies *HB* into *HB′*, and splits *HB′* into two totally ordered and contextualized sub-lists: HB'_1 with operations that are causally before O_x, and HB'_2 with operations that are concurrent with O_x, which can be achieved by using SV timestamps and a *transpose* procedure to reorder operations;
3. transforms O_x against operations in HB'_2, which can be expressed as $LT(O_x, HB'_2) = O_x\{HB'_2\}$;
4. executes $O_x\{HB'_2\}$; and
5. saves $O_x\{HB'_2\}$ at the end of *HB* and transposes it to the position determined by its total ordering in *HB*.

It is worth highlighting that *HB′* is created only for transforming O_x for execution, while *HB* always maintains operations that are totally ordered and contextualized.

CP2-avoidance validation

We can use the total ordering position of O_x to split HB'_2 into two sublists: $HB'_2 = HB'_{2\text{-}1} + HB'_{2\text{-}2}$, where $HB'_{2\text{-}1}$ contains operations totally before O_x and $HB'_{2\text{-}2}$ contains operations totally after O_x. Then it can be shown that $HB'_{2\text{-}1}$ is an OSP1 with respect to O_x; $HB'_{2\text{-}2}$ is an OSP2 with respect to O_x $\{HB'_{2\text{-}1}\} = LT(O_x, HB'_{2\text{-}1})$ by following similar reasoning as for the Jupiter case, except that the total ordering of operations is defined by sequence numbers and concurrent operations are detected by SVs.

One special complication in SOCT3 is the use of a transpose procedure to reorder operations in *HB*. This transpose procedure involves a pair of *reversible* inclusion/forward and exclusion/backward transformation functions [23], which is non-trivial to achieve [18, 21].

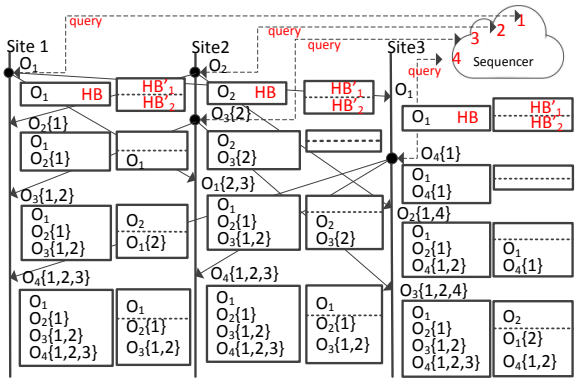

Figure 5: SOCT3: there is a central sequencer for issuing continuous numbers to totally order operations.

An example

Consider operation $O_3\{2\}$ at Site *2* in Figure 5. Upon generation, it is executed immediately, timestamped by a sequence number "*3*" and a SV, appended at the end of the local *HB*, and propagated to Site *1* and Site *3*. When $O_3\{2\}$ arrives at Site *3,* it waits until operations with sequence numbers smaller than "*3*" have been executed. Then, the original $HB = [O_1, O_2\{1\}, O_4\{1,2\}]$ is copied into *HB′* and split into two sub-list: $HB'_1 = [O_2]$, and $HB'_2 = [O_1\{2\}, O_4\{1,2\}]$ by using *SV* timestamps and the transpose procedure. $O_3\{2\}$ is transformed with concurrent operations in HB'_2 to get $O_3\{1,2,4\}$. The transformed operation $O_3\{1,2,4\}$ is executed, appended at the end of the original *HB*, and transposed to the position determined by the sequence number "*3*" to produce $HB = [O_1, O_2\{1\}, O_3\{1,2\}, O_4\{1,2,3\}]$.

SOCT4

SOCT4 follows SOCT3 in using a central sequencer to totally order operations, but eliminates the use of state-vectors

for capturing concurrency relations and the transpose procedure for reordering operations in *HB* by globally serializing operation propagation.

Local processing

When O_x is generated, SOCT4 handles it as follows:

1. executes O_x immediately and acquires a sequence number from the central sequencer to timestamp O_x;
2. saves O_x in *HB*, and waits until O_x has been symmetrically transformed with a sequence of (remote) concurrent operations, denoted as L_1, that are totally before O_x; this transformation can be expressed as: $SLT(O_x, L_1) = (O_x\{L_1\}, L_1\{O_x\})$;
3. propagates $O_x\{L_1\}$ (serialized operation propagation).

Remote processing

SOCT4 handles $O_x\{L_1\}$ at a remote site as follows:

1. waits until all operations that are totally before $O_x\{L_1\}$ have been executed at this site;
2. transforms $O_x\{L_1\}$ symmetrically with a sequence of local operations that are totally after it in *HB*, denoted as L_2; this transformation can be expressed as: $SLT(O_x\{L_1\}, L_2) = (O_x\{L_1, L_2\}, L_2\{O_x\})$;
3. executes $O_x\{L_1,L_2\}$ (no need to save it in the buffer).

CP2-avoidance validation

It can be shown that L_1 is an OSP1 with respect to O_x in $SLT(O_x, L_1)$, and L_2 is an OSP2 with respect to $O_x\{L_1\}$ in $SLT(O_x\{L_1\}, L_2)$ by the same reasoning as for Jupiter. By Theorem 3, SOCT4 CP2-avoidance is confirmed.

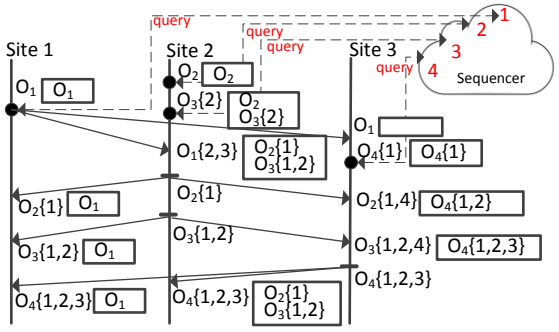

Figure 6: SOCT4: there is a central sequencer; each site maintains a buffer for locally generated operations only.

An example

Consider operation O_2 at Site *2* in Figure 6. Upon its generation, it is executed immediately at Site *2*, timestamped by a sequence number "*2*", and appended at the end of the local buffer. It is not propagated until it has been symmetrically transformed with O_1, which is the only operation totally before it. The transformed operation $O_2\{1\}$ is propagated to Site *1* and Site *3*. When $O_2\{1\}$ arrives at Site *3*, it is symmetrically transformed with $O_4\{1\}$ that is totally after it to produce two transformed operations: one is $O_4\{1,2\}$, which replaces $O_4\{1\}$ in the buffer; the other is $O_2\{1,4\}$, which is executed at Site *3*.

TIBOT

TIBOT (Time-Interval Based OT) is unique in using a distributed scheme to totally order operations, without using any central server. Collaborating sites may communicate with each other in any way that ensures reliability and the FIFO property.

Each site maintains a local timer (a numeric counter) with continuously increasing *Time-Interval* (*TI*) values: *0*, *1*, *2*, ..., etc. Timers at different sites must take the same sequence of values, but may tick at different speeds and need not be synchronized.

TI-based Timestamping Scheme: when an operation is generated at a site, it is timestamped by a tuple <*TI, SI, SN*>, where *TI* is the local timer value, *SI* is the local site identifier, *SN* is the local operation sequence number.

TI-based Total Ordering Scheme: for any two operations O_a and O_b, with timestamps T_a and T_b, respectively, $O_a \Rightarrow O_b$ *if and only if*: (1) $T_a.TI<T_b.TI$; or (2) $T_a.TI=T_b.TI$ and $T_a.SI < T_b.SI$; or (3) $T_a.TI=T_b.TI$ and $T_a.SI=T_b.SI$ and $T_a.SN<T_b.SN$.

Local processing

When O_x is generated, TIBOT handles it as follows:

1. executes O_x immediately and timestamps it with T_x;
2. saves O_x in a history buffer (*HB*) and waits until: (1) the local timer has been advanced to a value larger than $T_x.TI$; and (2) O_x has been transformed with a sequence of remote operations, denoted as L_1, which includes all operations whose *TI*s are smaller than $T_x.TI$; this can be expressed as $LT(O_x, L_1) = O_x\{L_1\}$;
3. propagates $O_x\{L_1\}$ with the same timestamp T_x.

Two important operation propagation schemes:

1. All local operations generated during the same time interval are packaged and propagated to remote sites together, which not only reduces communication overhead, but also simplifies derivation of total order relations at remote sites.
2. If there is no operation generated at a site during a time interval, this site is required to broadcast a special message, with a timestamp <*TI, SI, nil*>, at the end of each time interval. Such special messages are needed to enable the derivation of TI-based conditions and total ordering relations among operations, which are essential in TIBOT.

Remote processing

TIBOT handles $O_x\{L_1\}$ at a remote site as follows:

1. waits until: (1) the local timer has advanced to a value greater than $T_x.TI$; (2) local operations with *TI*s equal to the $T_x.TI$ have been propagated; and (3) all operations totally before $O_x\{L_1\}$ have been executed at this site;
2. undoes a sequence of (local) operations in *HB*, denoted as L_{undo}, which are totally after $O_x\{L_1\}$;
3. transforms $O_x\{L_1\}$ against a sequence of operations in *HB*, denoted as L_2, whose *TI*s are the same as $T_x.TI$; this can be expressed as $LT(O_x\{L_1\}, L_2) = O_x\{L_1, L_2\}$;
4. executes $O_x\{L_1, L_2\}$ and saves it in *HB*;

5. transforms operations in L_{undo} symmetrically with $O_x\{L_1, L_2\}$; this transformation can be expressed as $SLT(O_x\{L_1, L_2\}, L_{undo}) = (O_x\{L_1,L_2,L_{undo}\}, L_{undo}\{O_x\})$;
6. re-executes transformed operations in $L_{undo}\{O_x\{L_1, L_2\}\}$ and saves them sequentially in *HB*.

CP2-avoidance validation

It can be shown that L_1 is an OSP1 with respect to O_x in $LT(O_x, L_1) = O_x\{L_1\}$, L_2 is an OSP1 with respect to $O_x\{L_1\}$ in $LT(O_x\{L_1\}, L_2)$; and L_{undo} is an OSP2 with respect to $O_x\{L_1, L_2\}$ in $SLT(O_x\{L_1, L_2\}, L_{undo})$ by following the same reasoning as for Jupiter, except that the total ordering and concurrency relations among operations are detected by TI-based distributed timestamps. By Theorem 3, TIBOT CP2-avoidance is confirmed.

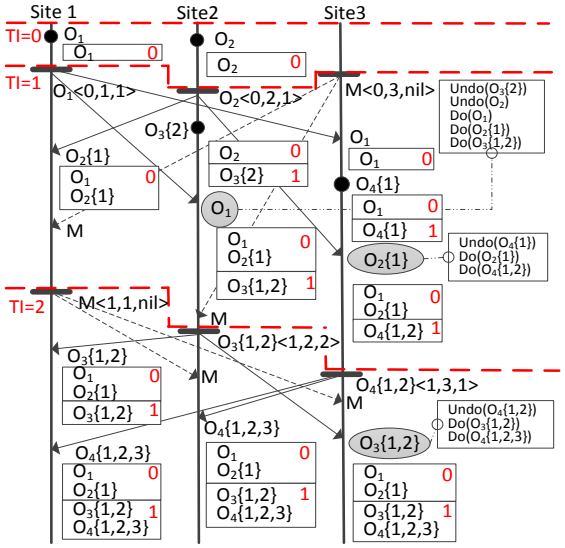

Figure 7: TIBOT: each site maintains an independent timer and a linear HB; operations may be undone and redone; special message M is sent if no local operation to propagate.

An example

Consider operation O_2 generated during $TI = 0$ at Site *2* in Figure 7. Upon generation, it is executed immediately, timestamped with $<0, 2, 1>$, and saved at the end of the *HB*. O_2 is not propagated until its timer has advanced to $TI = 1$. TIBOT handles O_2 at remote sites as follows:

1. At Site 1: Step 1, the following conditions are met: (1) local timer has advanced to $TI=1$; (2) local operation O_1, with the same $TI=0$ as O_2, has been propagated; and (3) O_1 is the only operation totally-before O_2 and has been executed. Step 2, there is no local operation totally-after O_1, so nothing to undo. Step 3, O_2 is transformed with O_1 to produce $O_2\{1\}$. Step 4, $O_2\{1\}$ is executed and saved in *HB*. There is nothing to re-transform and redo in Steps 5 and 6.
2. At Site 3: Step 1, the following conditions are met: (1) local timer has advanced to $TI=1$; (2) there is no local operation during $TI=0$ and a special timestamp message $M<0, 3, nil>$ has been propagated; and (3) O_1 is the only operation totally-before O_2 and has been executed. Step 2, local operation $O_4\{1\}$ is undone as it is totally after O_2. Step 3, O_2 is transformed with O_1 to produce $O_2\{1\}$. Step 4, $O_2\{1\}$ is executed and saved in *HB*. Step 5, $O_4\{1\}$ is transformed with $O_2\{1\}$ to produce $O_4\{1,2\}$. Step 6, $O_4\{1,2\}$ is re-executed and saved in *HB*. $O_4\{1,2\}$ is propagated when *TI* is advanced to *2*.

COT

Among seven OT systems, COT (Context-based OT) [21] is unique in supporting both consistency maintenance and *group undo* (initiated by users, not as internal concurrency control mechanisms) [11, 13, 21]. In this study, we focus on COT's consistency maintenance (convergence) aspects only. To avoid CP2, COT specifies a total ordering relation among operations and uses this relation to control operation execution and transformation orders, but leaves the schemes to achieve such ordering unspecified. In other words, COT may adopt a server-based or a distributed scheme that is able to meet the specified total ordering requirements. In the following, we sketch COT under the assumption that every collaborating site is connected to a central server via a TCP communication channel; the server broadcasts and serializes operations, but does not transform operations.

COT is also special in decoupling its operation buffering scheme from the core algorithm. The basic COT buffering scheme requires saving operations in original forms only. However, more sophisticated COT buffering schemes may selectively save operations in transformed forms to eliminate re-transformations [21]. For simplicity, we assume the basic buffering scheme in COT description.

Local processing

When an operation O_x is generated at a site, it is executed immediately and timestamped by a Context-Vector (CV) for capturing causal/context relations among operations, and propagated as-is to remote sites (via the server). All local operations are saved in original forms in a local buffer, called *Document State* (*DS*), which may be implemented as a set or other suitable data structures.

Server processing

The server broadcasts O_x to all sites, including the site which O_x comes from, which serves as a notification of its total ordering based on the server serialization.

Remote processing

COT handles O_x at a remote site as follows:

1. waits until all operations that are totally before O_x have been executed at this site;
2. finds all operations in the local *DS* that are concurrent with O_x by using their context-vectors;
3. contextualizes those operations into a sequence of operations, denoted as L, which are totally ordered according to the server serialization;
4. transforms O_x against operations in L, which can be expressed as: $LT(O_x, L)=O_x\{L\}$;
5. executes $O_x\{L\}$;
6. saves the original O_x in the local *DS*.

CP2-avoidance validation

We can use the total ordering position of O_x to split L into two sublists: $L = L_1 + L_2$, where L_1 contains operations totally before O_x and L_2 contains operations totally after O_x. Then, it can be shown that L_1 is an OSP1 with respect to O_x in $LT(O_x, L_1)$, and L_2 is an OSP2 with respect to $O_x\{L_1\}$ in $LT(O_x\{L_1\}, L_2)$ by following the same reasoning as for prior systems, except concurrency relations among operations are detected by context-vectors. By Theorem 3, COT CP2-avoidance is confirmed, which is consistent with prior verification based on the general CP2-avoidance condition [21].

An example

Consider operation $O_3\{2\}$ at Client *2* in Figure 8. Upon generation, it is executed immediately, saved in the local *DS*, and propagated to the server. The server broadcasts $O_3\{2\}$ to all clients. When $O_3\{2\}$ arrives at Client *3,* the local $DS=\{O_1, O_4\{1\}, O_2\}$. From *DS*, COT finds a subset of two operations $\{O_1, O_4\{1\}\}$ which are concurrent with $O_3\{2\}$, and converts the subset into a totally ordered and contextualized list $L= [O_1\{2\},O_4\{1,2\}]$. Then, $O_3\{2\}$ is transformed against L to get $O_3\{1,2,4\}$. Finally, $O_3\{1,2,4\}$ is executed on the local document, and original operation $O_3\{2\}$ is saved in the local $DS =\{O_1, O_4\{1\}, O_2, O_3\{2\}\}$.

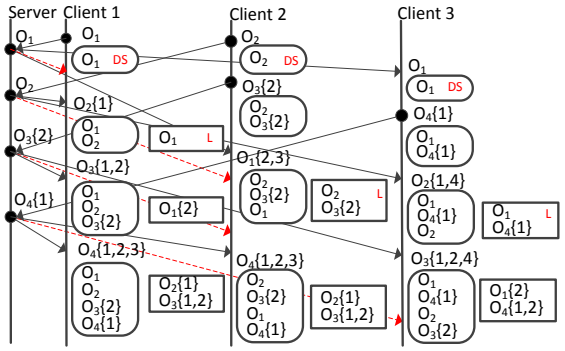

Figure 8: COT: every client maintains a document state *DS* containing executed operations in their original forms; concurrent operations are extracted from *DS* and contextualized in *L* in a totally ordered fashion for transformation.

Summary of CP2-avoidance Validation

CP2-avoidance validation results for seven OT systems are summarized in Table 1. As all transformations performed by these systems can be expressed by the basic CP2-avoiding operation sequences and transformation patterns established in this work, we have confirmed CP2-avoidance correctness for all systems:

1. COT was verified for its CP2-avoidance by using the general CP2-avoidance condition [21]; this work is a reconfirmation of its correctness based on the special CP2-avoidance condition.
2. Jupiter and Google OT were designed to solely use CP1-preserving transformation functions for convergence, which is essentially the same as what is meant by CP2-avoidance; this work establishes their correctness in achieving convergence without requiring transformation functions to preserve CP2.
3. NICE, TIBOT, and SOCT3/4 were designed to avoid CP2 but verified by using the insufficient condition. It is not surprising that these systems were able to meet the insufficient condition because they can satisfy the special CP2-avoidance condition (in this work), which satisfies the general CP2-avoidance condition (according to Theorem 2), which implies the insufficient CP2-avoidance condition, as proven in [21]. The value of this work is to establish their CP2-avoidance correctness on a sound ground.

Conversely, these validation results have also confirmed the generality and effectiveness of the basic CP2-avoidance conditions, mechanisms and transformation patterns, and validation framework, for studying and verifying a variety of OT systems.

OT Systems	Operation Sequences and Transformation Patterns at Three Locations		
	Local	**Server**	**Remote**
Jupiter OT	O_x	$SLT(O_x,L_1)$, L_1 *is an OSP1 with* O_x	$SLT(O_x\{L_1\},L_2)$ L_2 *is an OSP2with* $O_x\{L_1\}$
NICE	O_x	$SLT(O_x,L_1)$, L_1 *is an OSP1 with* O_x	$SLT(O_x\{L_1\},L_2)$, L_2 *is an OSP2 with* $O_x\{L_1\}$
Google OT	$SLT(O_x, L_1)$, L_1 *is an OSP1 with* O_x	$LT(O_x\{L_1\},L_2)$ L_2 *is an OSP1 with* $O_x\{L_1\}$	$SLT(O_x\{L_1,L_2\}, L_3)$, L_3 *is an OSP2 with* $O_x\{L_1,L_2\}$
SOCT3	O_x	*Sequencer*	$SLT(O_x, HB'_2)$, $HB'_2=HB'_{21}+HB'_{22}$, HB'_{21} *is an OSP1with* O_x HB'_{22} *is an OSP2 with* $O_x\{HB'_{21}\}$
SOCT4	$SLT(O_x, L_1)$, L_1 *is an OSP1 with* O_x	*Sequencer*	$SLT(O_x\{L_1\},L_2)$, L_2 *is an OSP2 with* $O_x\{L_1\}$
TIBOT	$LT(O_x, L_1)$, L_1 *is an OSP1 with* O_x	*N.A.*	$LT(O_x\{L_1\}, L_2)$, L_2 *is an OSP1 with* $O_x\{L_1\}$; $SLT(O_x\{L_1, L_2\}, L_{undo})$, L_{undo} *is an OSP2 with* $O_x\{L_1,L_2\}$
COT	O_x	*Serialization and broadcast*	$LT(O_x, L)$, $L=L_1+L_2$, L_1 *is an OSP1 with* O_x, L_2 *is an OSP2 with* $O_x\{L_1\}$

Table 1: CP2-avoidance validation summary

NEW SYSTEMS/IDEAS INSPIRED BY THIS STUDY

This study has not only enabled us to discover the basic conditions and transformation patterns for CP2-avoidance and to validate existing OT systems, but also inspired us to identify improvements to these OT systems, devise new OT systems capable of avoiding CP2 with unique combinations of novel features, and ideas for future exploration.

Improvements to Jupiter and Google OT

From NICE, we learned 1D buffers are adequate to achieve CP2-avoidance with simple control algorithms. By comparing Jupiter with NICE, we observed their CP2-avoiding operation sequences and transformation patterns and locations are the same (as shown in Table 1), and their main differences lie in the use of 2D state-spaces (in Jupiter) vs. 1D buffers (in NICE). From this observation, we derive

Jupiter can be simplified by replacing all 2D state-spaces with 1D buffers similar to NICE, without losing any capability or complicating any aspect of Jupiter.

Google OT has replaced the multiple 2D state-spaces in Jupiter server with a single 1D buffer, at the price of a *stop-and-wait* synchronization between each client and the server, but still maintains 2D state-spaces at client sites. Based on the same NICE insight, Google OT can also be simplified by replacing 2D spaces with 1D buffers.

A New OT System: TIBOT 2.0

Compared to other six OT systems, TIBOT is unique not only in its distributed total ordering scheme, but also in its capability of avoiding both CP1 and CP2, which is only achieved by the prior GOT system [18]. But TIBOT avoids both CP1 and CP2 without using state-vectors or exclusive transformation, which is an advantage over GOT. Similar to GOT [18], TIBOT has used an *undo/transform-do/transform-redo* scheme to enforce a totally ordered transformation among all operations and never transform a pair of operations in different orders. However, past research has found undo/redo-based concurrency control schemes are inefficient and may cause interface abnormality to end-users, so most OT systems have been designed to avoid internal undo/redo.

Inspired by the insights drawn from this study, we propose a new OT system, named as TIBOT 2.0, which inherits from TIBOT the basic time-interval-based distributed total ordering and shares the same local processing and propagation control schemes as TIBOT, but has a significantly re-signed remote processing to eliminate undo/redo. We describe TIBOT 2.0 remote processing below.

Remote processing in TIBOT 2.0

TIBOT 2.0 handles $O_x\{L_1\}$ at a remote site as follows:

1. waits until: the same conditions as specified in Step 1 in the original TIBOT;
2. transforms $O_x\{L_1\}$ against a sequence of operations in *HB*, denoted as L_2, which includes all operations with *TI*s equal to $T_x.TI$ and totally before $O_x\{L_1\}$; this transformation can be expressed as $LT(O_x\{L_1\}, L_2) = O_x\{L_1, L_2\}$;
3. saves $O_x\{L_1, L_2\}$ at the end of L_2 in *HB*;
4. transforms $O_x\{L_1, L_2\}$ symmetrically with a sequence of operations in *HB*, denoted as L_3, which are totally after $O_x\{L_1, L_2\}$; this transformation can be expressed as $SLT(O_x\{L_1, L_2\}, L_3) = (O_x\{L_1, L_2, L_3\}, L_3\{O_x\})$;
5. executes $O_x\{L_1,L_2,L_3\}$; replaces L_3 with $L_3\{O_x\}$ in *HB*.

The key insight in TIBOT 2.0 is (in Step 4) to transform $O_x\{L_1, L_2\}$ symmetrically with a sequence of operations in L_3, which are totally after $O_x\{L_1, L_2\}$. This symmetric transformation replaces undoing and redoing the same sequence of operations in L_{undo} (= L_3) in Steps 2 and 5 of the original TIBOT. This is the key to avoiding CP2 without undo/redo (but giving up CP1-avoidance as CP1 can be easily achieved by transformation functions).

In addition, TIBOT 2.0 made other subtle but important technical changes (due to the elimination of undo/redo):

- Step 2: the condition for transformation of $O_x\{L_1\}$ has been enhanced to require operations in L_2 to be totally before $O_x\{L_1\}$ as well as having *TI*s equal to T_x, which is in contrast to requiring *TI*s equal to T_x only in Step 3 in the original TIBOT.
- Step 5: $O_x\{L_1, L_2, L_3\}$ is executed to achieve the original effect of O_x, which is in contrast to executing $O_x\{L_1, L_2\}$ in Step 4 and re-executing undone operations in Step 6 in the original TIBOT.

TIBOT 2.0 is unique in avoiding CP2 by distributed total ordering (no central server) without undo/redo. It can be shown that L_2 is OSP1 with respect to $O_x\{L_1\}$ in $LT(O_x\{L_1\}, L_2)$, and L_3 is OSP2 with respect to $O_x\{L_1, L_2\}$ in $SLT(O_x\{L_1,L_2\}, L_3)$ by following the same reasoning as in validating other systems.

In Figure 9, we illustrate the working of TIBOT 2.0 under the same working example. Consider O_2 at Site *2*, local processing of O_2 is the same as in the original TIBOT. TIBOT 2.0 handles O_2 at Site *3*, where $HB = [O_1, O_4\{1\}]$, as follows. Step 1, the same conditions are met. In Step 2, O_2 is transformed with $L_2 = [O_1]$, which is totally before O_2, to produce $O_2\{1\}$. Step 3, $O_2\{1\}$ is saved in *HB* after O_1. Step 4, $O_2\{1\}$ is symmetrically transformed with L_3=$[O_4\{1\}]$, which is totally after $O_2\{1\}$, to produce $O_2\{1,4\}$ and $O_4\{1,2\}$. Step 5, $O_2\{1,4\}$ is executed at Site *3*, and $O_4\{1,2\}$ is saved to replace $O_4\{1\}$ in *HB*. By comparing Figure 9 with Figure 7, we can see the undo/redo processes for O_1, $O_2\{1\}$ and $O_3\{1,2\}$ have been eliminated, and their execution forms have also been changed due to changes in Step 5 of TIBOT 2.0.

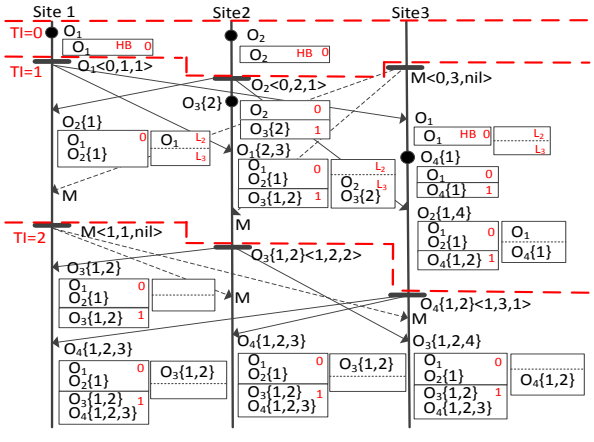

Figure 9: TIBOT 2.0: achieving CP2-avoidance without undo/transform-do/transform-redo or a central server.

Seeking Alternative CP2-avoidance Conditions

From this study, we have found all existing systems capable of avoiding CP2 meet the same special CP2-avoidance condition as specified in Definition 6. However, this is only one possible specialization of the general condition, which requires operations in the context of two operations to be the same whenever they are transformed with each other.

One interesting question arises: are there other possible specializations of this general CP2-avoidance condition? The answer is *yes*.

One alternative specialization is to restrict operations in the context of O_a and O_b to be totally before both O_a and O_b only, i.e. $O_x \Rightarrow O_a \Rightarrow O_b$, (keeping Condition 2-1 in Definition 6), but disallow operations that are totally ordered between O_a and O_b (removing Condition 2-2 in Definition 6). This specialization is valid as it still ensures the context uniqueness of two operations whenever they are transformed and meets other established context-based conditions [21]. However, careful examination reveals this specialization would prohibit users from generating operations continuously, which is undesirable.

Seeking for other valid and desirable special CP2-avoidance conditions (e.g. to get rid of totally ordering operations) is an interesting direction for further research as it may inspire invention of novel OT systems for meeting special algorithmic or application needs in the future.

CONCLUSION

In this work, we have conducted a comprehensive and in-depth study on convergence property preservation and avoidance in OT systems in general, and in validating CP2-avoidance for seven major OT systems: Jupiter, NICE, Google OT, SOCT3, SOCT4, TIBOT, and COT. From this study, we have made three major contributions: (1) establishment of CP2-avoidance correctness of seven major OT systems; (2) discovery of CP2-avoidance conditions, operation sequence and transformation patterns, basic mechanisms and a general framework for studying a diverse range of OT systems; and (3) improvements to existing systems (Jupiter and Google OT for Wave and Docs), and design of a new system TIBOT 2.0 – the first OT system based on a distributed total ordering scheme and capable of avoiding CP2 without using undo/redo. These results have significantly contributed to the advancement of OT knowledge and technique, and collaboration-enabling technology in general.

Inspired by the insights from this study, we are designing new OT systems to support both consistency maintenance and group undo, without using vector-based schemes, in large scale collaborations with an arbitrary number of dynamic collaborating users. Other major efforts in our group are devoted to applying OT technologies to advanced real-world collaborative applications.

ACKNOWLEDGEMENT

This research is partially supported by an Academic Research Fund Tier 1 Grant from Ministry of Education Singapore, and by NTU NAP Grant M4080738.020. The authors wish to thank anonymous reviewers for their insightful and constructive comments and suggestions.

REFERENCES

1. Agustina and Sun, C. Dependency-conflict detection in real-time collaborative 3D design systems. *ACM CSCW* (2013), 715 – 728.
2. Begole, J., Rosson, M.B. and Shaffer, C.A. Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems. *ACM TOCHI* 6, 2 (1999), 95 – 132.
3. Ellis, C. A. and Gibbs, S. J. Concurrency control in groupware systems. *ACM SIGMOD* (1989), 399–407.
4. Greenberg, S. and Marwood, D. Real time groupware as distributed system: concurrency control and its effect on the interface. *ACM CSCW* (1994), 207 – 217.
5. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *CACM 21*, 7 (1978), 558-565.
6. Li, R. and Li, D. A landmark-based transformation approach to concurrency control in group editors. *ACM GROUP* (2005), 284 – 293.
7. Li, D. and Li, R. An admissibility-based operational transformation framework for collaborative editing systems. *JCSCW 19, 1* (2010): 1 – 43.
8. Li, R., Li, D. and Sun, C. A time interval based consistency control algorithm for interactive groupware applications. *IEEE ICPADS* (2004), 429 – 436.
9. MacFadden, M. The client stop and wait operational transformation control algorithm. *Solute Consulting,* San Diego, CA, 2013.
10. Nichols, D., Curtis, P., Dixon, M. and Lamping, J. High-latency, low-bandwidth windowing in the Jupiter collaboration system. *ACM UIST* (1995), 111-120.
11. Prakash, A. and Knister, M. A framework for undoing actions in collaborative systems. *ACM TOCHI 1, 4* (1994), 295 – 330.
12. Oster, G., Molli, P., Urso, P. and Imine, A. Tombstone transformation functions for ensuring consistency in collaborative editing systems. *IEEE CollaborateCom* (2006).
13. Ressel, M. and Gunzenhauser, R. Reducing the problems of group undo. *ACM GROUP* (1999), 131–139.
14. Ressel, M., Ruhland, N. and Gunzenhauser, R. An integrating, transformation-oriented approach to concurrency control and undo in group editors.*ACM CSCW* (1996),288 – 297.
15. Shen, H.F. and Sun, C. Flexible notification for collaborative systems. *ACM CSCW* (2002), 77 – 86.
16. Spiewak, D. Understanding and applying operational transformation. http://www.codecommit.com/blog/ java/understanding-and-applying-operational-transformation.
17. Sun, C. and Ellis, C. Operational transformation in real-time group editors: issues, algorithms, and achievements. *ACM CSCW* (1998), 59 – 68.
18. Sun, C., Jia, X., Zhang, Y., Yang, Y., and Chen, D. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. *ACM TOCHI 5,1* (1998), 63 – 108.
19. Sun, C., Xia, S., Sun, D., Chen, D., Shen, H. and Cai, W. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM TOCHI 13, 4* (2006), 531 – 582.

20. Sun, C. OTFAQ: operational transformation frequent asked questions. http://cooffice.ntu.edu.sg/otfaq.
21. Sun, D. and Sun, C. Context-based operational transformation in distributed collaborative editing systems. *IEEE TPDS* (2009), 1454 – 1470.
22. Sun, C. Wen, H. and Fan, H. Operational transformation for orthogonal conflict resolution in collaborative two-dimensional document editing systems. *ACM CSCW* (2012), 1391 – 1400.
23. Vidot, N., Cart, M., Ferrie, J. and Suleiman, M. Copies convergence in a distributed real-time collaborative environment. *ACM CSCW* (2000), 171 – 180.
24. Wang, D., Mah, A. and Lassen, S. Google wave operational transformation. *http://www.waveprotocol.org/whitepapers/operational-transform.*
25. Xu, Y., Agustina, and Sun, C. Exhaustive search of puzzles in operational transformation. *ACM CSCW* (2014).