Capturing Causality by Compressed Vector Clock in Real-time Group Editors

Chengzheng Sun
School of Computing and Information Tech.
Griffith University
Brisbane, Qld 4111, Australia
C.Sun@cit.gu.edu.au
http://www.cit.gu.edu.au/∼scz

Wentong Cai
School of Computer Engineering
Nanyang Technological University
Singapore 639798
aswtcai@ntu.edu.sg
http://www.ntu.edu.sg/home/aswtcai

Abstract

Real-time group editors allow a group of users to view and edit the same document at the same time over the Internet. They are a special class of distributed applications in the area of groupware. Vector logical clock is a powerful technique to capture causality in distributed computing systems. In general, the minimum size of a vector clock is N — the number of communicating processes in a distributed system. In this paper, we propose a novel technique to compress the vector size from N to a constant 2 by means of operational transformation — an innovative technique invented by groupware research for consistency maintenance in real-time group editors. We will show how compressed vector clocks can be used as an effective and efficient means for operation timestamping and concurrency detection in group editors. The proposed technique has been implemented in a web-based real-time group editor which allows an arbitrary number of users to participate a collaborative editing session. The basic ideas and techniques of this work may be generalized and potentially applicable to other distributed computing systems and applications.

Keywords: vector logical clock, operational transformation, real-time group editors, distributed systems, Web-based applications.

1 Introduction

Vector logical clock is a powerful technique to capture causality in distributed computing systems and applications [6, 10, 12]. It has been used in a range of distributed systems, including group communication [1], distributed debugging and monitoring [2, 12], distributed and parallel simulation [3, 11], and groupware systems [5, 14]. However, vector clock has a major drawback: the vector size is N – the number of communicating processes in a distributed system. When N is very large, the amount of timestamp data (communication overhead) that has to be attached to each message may become unacceptable.

Various methods have been proposed to compress the size of vector clocks. At one extreme, methods have been proposed in [7, 12] to reduce the timestamp data to a single integer (i.e., a scalar instead of a vector of N elements), but these methods rely on the availability of a complete dependency graph for all events in a distributed computation in order to calculate the full vector time. The computational overhead for calculating the vector time for each event can be too large for an on-line computation and may slow down the distributed computation in an unacceptable way. Therefore, these methods are mainly applicable for trace-based off-line analysis of causality in a distributed computation.

Other methods have been proposed to dynamically compress the size of vector clocks [9, 13]. These methods are all based on the following observation: even though the number N of processes is large, only few of them are likely to interact frequently by direct message exchanges. So there is no need to carry a full vector of N elements in each message whenever there is a communication between a pair of processes. The basic idea is to carry in each message only the necessary new elements of a vector which have been updated since the previous communication between any pair of processes. Based on the information in the compressed vector and the additional information about the local vector clock in the receiving process, the original full vector can be recovered, so causality can be captured. These methods can substantially reduce the communication overhead in large distributed computations. However, the main problem with this approach is that the size of the message timestamps is still linear in N in the worst case.

It has been proven that the causality relationship among N communicating processes has in general *dimension* N, which induces a lower bound on the size of vector clocks [4]. It seems impossible to reduce the vector clock size to a value less than N if the causality relationship among communi-

Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)
1530-2075/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:19:02 UTC from IEEE Xplore. Restrictions apply.

cating processes remains N dimensions. However, if it was possible to transform the dimension of the causality relationship among messages from N to a smaller dimension, then it would be possible to reduce the vector clock size from N to a smaller size. In this paper, we propose a novel vector clock compressing technique with such a capability. Our approach is able to compress the vector clock to a constant size of 2, regardless of the number N of communicating processes in the system. This approach was motivated by our research in the area of real-time group editors [14].

Real-time group editors allow a group of users to view and edit the same document at the same time over the Internet [5, 14]. They are a special class of distributed applications in the areas of groupware or CSCW (Computer-Supported Cooperative Work). Capturing causality among editing operations is essential for consistency maintenance in real-time group editors [14]. The basic idea of our approach is to transform the N-dimensional causality relationship among N communicating processes into a 2-dimension causality relationship, by means of a novel concurrency control technique invented by groupware research, called operational transformation [15]. We will show how compressed vector clock can be used as an effective and efficient means for operation timestamping and concurrency detection in group editors.

The rest of this paper is organized as follows. First, some background information about real-time group editors is given in Section 2. Then, a compressed vector clock scheme is proposed for operation timestamping in Section 3. The concurrency checking scheme based on the compressed vector clock is presented in Section 4. An example for illustrating the timestamping and concurrency checking schmes is given in Section 5. Finally, major contributions of this work are summarized in Section 6.

2 Issues in group editors

The goal of our research is to design and implement a real-time group editor meeting the following requirements [14]: (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) *high 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. 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 the local interface immediately. Afterwards, a local operation must be propagated to remote collaborating sites for execution in order to maintain consistency [14].

2.1 Star-like communication topology

Most real-time group editors using a replicated architecture are also fully *distributed*, i.e., all collaborating sites communicate with each other directly for propagating their local operations. Example systems include the GROVE system [5], and the REDUCE (REal-time Distributed Unconstrained Cooperative Editing) system [14]. The REDUCE system was originally implemented as a stand-alone Java application running on multiple collaborating sites connected by the Internet.

The World Wide Web provides an excellent framework for building and using collaborative systems. To integrate the REDUCE system into the Web framework, we decided to convert each REDUCE collaborating site into a Java applet running inside a Web browser (e.g., the Netscape or Internet Explorer). Because of the security restrictions imposed by Web browsers, a Java applet can only communicate directly with the Web server machine from which it is loaded. Therefore, it is not possible for REDUCE editing applets to communicate directly for propagating editing operations.

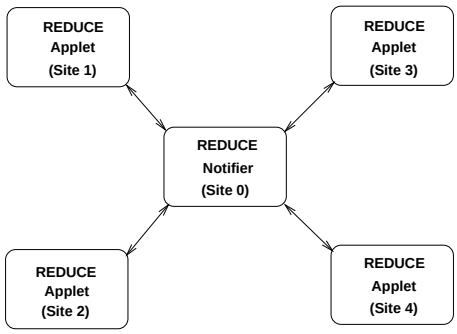

Figure 1: Star-like topology of Web-based REDUCE

To facilitate the propagation of editing operations, a notifier process, implemented as a Java application running at the Web server machine, has to be introduced. Like all REDUCE applets, the REDUCE notifier also maintains a full copy of the shared document, and executes all editing operations generated by any collaborating sites. A local editing operation is executed by the REDUCE applet immediately after its generation, giving the quickest response to the user. Then this operation is propagated to the notifier site, which will execute this operation on its copy of the shared document and then broadcast it to all other REDUCE applets except the one the operation comes from. In effect, the notifier site maps the N-way communication among N sites into a 2-way communication between itself and a collaborating site. The star-like communication topology of the Web-based REDUCE system is shown in Fig. 1, with the REDUCE notifier (Site 0) at the center of the star.

2.2 Inconsistency problems

A significant challenge in designing group editors with replicated documents is consistency maintenance. In a system

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:19:02 UTC from IEEE Xplore. Restrictions apply.

with a star-like topology and the FIFO property of TCP connections between the central notifier and collaborating sites, operations are guaranteed to arrive at all sites in their correct causal orders, thus avoiding the so-called causality violation problem [14]. However, other inconsistency problems still remain.

To illustrate, consider a collaborative editing scenario with four collaborating sites (including the notifier site (S_0)), as shown in Fig.2. Four operations are generated (denoted as bullets) in this scenario: O_1 at site 1, O_2 and O_3 at site 2, and O_4 at site 3. 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 0, for example, O_2 is executed first, followed by O_1, O_4, and O_3. After executing each operation at site 0, this operation is propagated in its *original* form to all other sites except the one from which the operation comes. In the absence of any consistency maintenance measure, two inconsistency problems manifest themselves in this scenario.

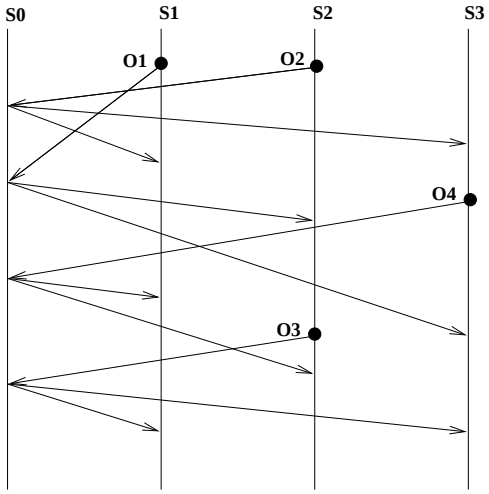

Figure 2: A scenario of a collaborative editing scenario

First, operations may arrive and be executed in different orders at different sites, resulting in different final results. This problem is called *divergence* [14]. As shown in Fig. 2, the four operations are executed in the following orders: O_2, O_1, O_4, and O_3 at site 0; O_1, O_2, O_4, and O_3 at site 1; O_2, O_1, O_3, and O_4 at site 2; and O_2, O_4, O_1, and O_3 at site 3. Unless operations are commutative (which is generally not the case), final editing results would not be identical among collaborating sites.

Second, due to concurrent generation of operations, the *actual effect* of an operation at the time of its execution may be different from the *intention* of this operation at the time of its generation. The intention of an operation is the execution effect which can be achieved by applying this operation on the document state from which it was generated [14]. This problem is called *intention violation* [14]. As shown in Fig. 2, operation O_1 is generated at site 1 without any knowledge of O_2 generated at site 2, so O_1 is *concurrent* with (or *independent* of) O_2, and vice versa (a precise definition about *concurrency* is given in the next subsection). At site 1, when O_2 is propagated from site 0, the document state at site 1 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, resulting in an editing effect which is different from the *intention* of O_2.

For example, assume the shared document initially contains the following sequence of characters: "ABCDE". Suppose $O_1 = Insert[\text{"12"}, 1]$, which intends to insert string "12" at position 1, i.e., between "A" and "BCDE"; and $O_2 = Delete[3, 2]$, which intends to delete three characters starting from position 2, i.e., "CDE". After the execution of these two operations, the *intention-preserved* result (at all sites) should be: "A12B". However, the actual result at site 1, obtained by executing O_1 followed by executing O_2, would be: "A1DE", which apparently violates the intention of O_1 since the character "2", which was intended to be inserted, is missing in the final text, and also violates the intention of O_2 since characters "DE", which were intended to be deleted, are still present in the final text.

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 [14].

2.3 Operational transformation

Operational transformation is a technique capable of resolving both divergence and intention-violation problems [5, 14, 15]. The basic idea of operational transformation is to reformulate the parameters of an operation according to the impact of precedingly executed *concurrent* operations, so that the operation becomes locally redefined but still retains its original effect.

To illustrate, consider the previous example. When O_2 arrives at site 1, it will not be executed in its original form, but first transformed against the concurrent operation O_1 to produce $O_2' = Delete[3, 4]$. O_2' is the form that O_2 would have taken if it was generated after the execution of O_1 at site 1. Executing O_2' in the new document state ("A12BCDE") would produce the same result ("A12B") as executing O_2 in the old document state ("ABCDE").

The novelty of operational transformation is that it allows concurrent operations to be executed in any orders, but ensures their final effects be identical and intention-preserved. To facilitate operational transformation, every site needs to maintain a *History Buffer (HB)* for saving executed operations. Local operations can be executed without transformation, but a remote operation must be transformed against concurrent operations in the HB before its execution. For detailed discussion of the operational transformation technique,

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:19:02 UTC from IEEE Xplore. Restrictions apply.

the reader is referred to [14, 15]. There is one point about operational transformation which is directly relevant to the topic of this paper: an operation needs to be transformed against *concurrent* operations only. Therefore, for operational transformation to work, it is essential to have a knowledge of causality relationship among operations.

2.4 Causality relationship of operations

We are interested in capturing the causality relationship among editing operations only. Following Lamport [8], we define a causal ordering relation on operations in terms of their generation and execution sequences.

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

Based on the causality relation, the concurrency relationship among operations can be derived.

Definition 2 *Concurrent operations*
Given two operations O_a and O_b. they are concurrent, expressed as $O_a \parallel O_b$, *iff* $O_a \not\rightarrow O_b$, and $O_b \not\rightarrow O_a$. □

For example, there are three pairs of causally related operations in Fig.2: $O_1 \rightarrow O_3$, $O_2 \rightarrow O_3$, and $O_2 \rightarrow O_4$ because the execution of O_1 happens before the generation of O_3, the generation of O_2 happens before the generation of O_3, and the execution of O_2 happens before the generation of O_4. Moreover, there are three pairs of concurrent operations: $O_1 \| O_2$, $O_1 \| O_4$, and $O_3 \| O_4$ because for any pair, neither operation's execution happens before the other operation's generation.

3 Compressed vector clock

3.1 Basic ideas

Most group editors have used a full vector clock of N elements for capturing causality relationship of operations among N collaborating sites [5, 14]. However, vector clocks can be compressed by taking advantage of the operational transformation technique, which is able to redefine a remote operation according to the local document state, and consequently redefine the causality relationship among operations.

The basic idea of our approach is to let the central notifier not only map between N-way communication and 2-way communication, but also convert between N-dimension causality relationship and 2-dimension causality relationship: for every operation received from a remote collaborating site, site 0 first transforms it against previously arrived concurrent operations, and then propagates the transformed operation, rather than the original operation, to other sites. In this way, a remote operation is effectively redefined on the current document state at site 0, so that it can be treated by other sites as if it was an operation directly generated by site 0. In other words, with the transformation effect at site 0, all operations propagated from site 0, regardless from which sites they were originated, have no difference from the operations really originated from site 0. It is this redefining effect on propagated operations which converts the N-dimension causality relationship into a 2-dimension causality relationship, and provides the opportunity to compress the vector clock size from N to 2.

3.2 Key data structures

For a system of N collaborating sites, each site has a unique identifier $i \in \{1, 2, ..., N\}$, and the additional notifier site is identified as site 0. Each site maintains a vector clock, called *State Vector (SV)*. In the following discussions, the terms of vector clock and state vector are interchangeable.

The state vector at site $i \neq 0$, denoted as SV_i, is a *compressed* vector of two elements, with $SV_i[1]$ for counting the total number of operations received from site 0, and $SV_i[2]$ for counting the number of operations generated by the local site. SV_i is maintained as follows:

1. Initially, $SV_i[j] := 0$, $1 \leq j \leq 2$.
2. After executing an operation propagated from site 0, $SV_i[1] := SV_i[1] + 1$.
3. After executing a local operation, $SV_i[2] := SV_i[2] + 1$.

However, the state vector at site 0, denoted as SV_0, is a *full* vector of N elements, with $SV_0[i]$ for counting the number of operation received from site i, $1 \leq i \leq N$. SV_0 is maintained as follows:

1. Initially, $SV_0[i] := 0$, $1 \leq i \leq N$.
2. After executing an operation received from site i, $SV_0[i] := SV_0[i] + 1$.

The reason for using a full state vector at site 0 is that site 0 communicates directly with N collaborating sites and it needs N elements to keep track the numbers of operations propagated from each of the N sites. However, the full state vector is only used locally at site 0 and is never used for timestamping propagated operations, as will be explained in the next section.

3.3 Operation timestamping scheme

When an operation is to be propagated to a remote site, or to be saved in the local *HB*, it must be timestamped. The timestamp of operation O is denoted as T_O.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:19:02 UTC from IEEE Xplore. Restrictions apply.

Timestamping propagated operations

The timestamp for any operation O propagated between site 0 and site $i \neq 0$ is always a compressed state vector of two elements.

At site $i \neq 0$, after executing a local operation O, the current value of the 2-element state vector is directly used to timestamp O, i.e., $T_O = SV_i$. Then, the timestamped O is propagated to site 0. In this case, $T_O[1]$ represents the total number of operations received from site 0, and $T_O[2]$ represents the number of operations generated by site i.

The situation at site 0 is different. After a remote operation O is received, it is first transformed against all concurrent operations in HB_0 and executed. Then, the transformed O, denoted as O', is propagated to all sites except the site from which O comes. To timestamp O', the N-element full state vector must be compressed to a 2-element state vector. Suppose O' is to be propagated to site i, the 2 elements of $T_{O'}$ are calculated as follows:

$$T_{O'}[1] = \sum_{j=1, j\neq i}^{N} SV_0[j], \quad (1)$$

$$T_{O'}[2] = SV_0[i], \quad (2)$$

where $T_{O'}[1]$ represents the total number of operations received from all sites except site i, and $T_{O'}[2]$ represents the number of operations received from site i.

This compression is valid because the propagated operation O' has been redefined at site 0, all operations propagated from site 0 to the destination site i can be counted as operations *virtually* generated at site 0, without the need to differentiate which operation was *originally* generated at which collaborating site. For the purpose of capturing the causality relationship between O' and other operations originated at site i, it is sufficient to know how many operations have been propagated from site 0 to site i (recorded in $T_{O'}[1]$), and how many operations have been received by site 0 from site i (recorded in $T_{O'}[2]$) at the time of propagating O'.

From the way $T_{O'}$ is calculated, we can see that the same O' is timestamped by a different compressed state vector for a different destination site i. In other words, different 2-element state vectors have to be calculated from the same N-element state vector, which explains why a full state vector has to be maintained at site 0.

Timestamping buffered operations

When an executed operation is saved in the HB, it must be timestamped as well.

At site $i \neq 0$, a buffered operation is timestamped with its original 2-element propagation timestamp.

At site 0, a buffered operation is timestamped with the current N-element state vector value. In other words, the 2-element propagation timestamp for a received operation has to be replaced with a N-element state vector. This is another reason why a full state vector has to be maintained at site 0. This full state vector timestamping is needed because the knowledge of exactly how many operations received from each site is essential for the purpose of operation concurrency check at site 0 (see the next section).

4 Concurrency checking scheme

If full state vectors were used to capture the causality relationship among operations, the following well-known property of vector clocks could be used to check the concurrency relationship between any pair of operations [10]: Given two operations O_a and O_b, generated at sites x and y and timestamped by T_{O_a} and T_{O_b}, respectively, we have:

$$O_a \parallel O_b \iff T_{O_a}[x] > T_{O_b}[x] \text{ and } T_{O_b}[y] > T_{O_a}[y], \quad (3)$$

where the symbol "$\iff$" means "*if and only if*".

In our system, concurrency check happens between a newly arrived remote operation and a previously executed operation in the HB, so both compressed state vectors (for the newly arrived remote operations) and full state vectors (for buffered operations at site 0) may be examined in this check. Therefore, it is necessary to adjust formula (3) in order to cope with the mixture of full and compressed state vectors. In addition, operations are guaranteed to arrive at every site in their right causal orders due to the use of the star-like communication topology and the FIFO property of TCP connections, so we can take advantage of this to simplify the concurrency check.

4.1 Concurrency check at site $i \neq 0$

Suppose O_a is an operation propagated from site 0, with a 2-element timestamp T_{O_a}, and O_b is an executed operation saved in the HB_i, with a 2-element timestamp T_{O_b}. Since timestamps for both operations are two-element compressed state vectors, it is straightforward to derive the following formula from formula (3):

$$O_a \parallel O_b \iff T_{O_a}[1] > T_{O_b}[1] \text{ and } T_{O_b}[y] > T_{O_a}[y], \quad (4)$$

where $y = 1$ if O_b was an operation propagated from site 0; or $y = 2$ if O_b was an operation generated at site i. $T_{O_a}[1] > T_{O_b}[1]$ implies that $O_a \not\rightarrow O_b$, and $T_{O_b}[y] > T_{O_a}[y]$ implies $O_b \not\rightarrow O_a$.

However, it is known that O_b has been executed before O_a's arrival, so $O_a \not\rightarrow O_b$ is guaranteed due to the star-like communication topology and the FIFO property of TCP connections. Therefore, condition $T_{O_a}[1] > T_{O_b}[1]$ is always satisfied and never needs to be checked. Consequently, formula (4) can be simplified as:

$$O_a \parallel O_b \iff T_{O_b}[y] > T_{O_a}[y]. \quad (5)$$

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:19:02 UTC from IEEE Xplore. Restrictions apply.

4.2 Concurrency check at site 0

Suppose O_a is an operation just propagated from a remote site, with a 2-element timestamp T_{O_a}, and O_b is an executed operation saved in the HB_0, with an N-element timestamp T_{O_b}. Suppose O_a and O_b were originally generated at sites x and y, respectively. To check whether O_a and O_b are concurrent, the N-element full state vector T_{O_b} has to be compressed into a proper 2-element state vector in order to become comparable with the 2-element state vector T_{O_a}. Generally, we have

$$\begin{aligned} O_a \parallel O_b \iff & T_{O_a}[2] > T_{O_b}[x] \ \textbf{and} \\ & (x = y \ \textbf{and} \ T_{O_b}[y] > T_{O_a}[2] \ \textbf{or} \\ & x \neq y \ \textbf{and} \ (\sum_{j=1, j\neq x}^{N} T_{O_b}[j]) > T_{O_a}[1]) \end{aligned} \quad (6)$$

The correctness of formula (6) can be verified as follows. First, $O_a \not\rightarrow O_b$ is implied by $T_{O_a}[2] > T_{O_b}[x]$, according to the first condition of formula (3) and the timestamping scheme at site $i \neq 0$. Second, $O_b \not\rightarrow O_a$ is implied by $T_{O_b}[y] > T_{O_a}[2]$ if $x = y$, or by $(\sum_{j=1,j\neq x}^{N} T_{O_b}[j]) > T_{O_a}[1]$ if $x \neq y$, according to the second condition of formula (3) and the timestamping scheme for buffered operations at site 0. The compression for T_{O_b} in formula (6) is corresponding to the compression formula (1) used at the time when O_b was propagated to remote sites. Therefore, it is valid for the same reason as explained before.

From the way T_{O_b} is compressed, it can be seen that the same buffered operation timestamp T_{O_b} is compressed into different 2-element state vectors for operations from different remote sites. This is the reason why the full state vector has to be used in timestamping buffered operations at site 0.

Since it is known that O_b has been executed before O_a's arrival, $O_a \not\rightarrow O_b$ is guaranteed due to the star-like communication topology and the FIFO property of TCP connections. Therefore, condition $T_{O_a}[2] > T_{O_b}[x]$ is always satisfied and never needs to be checked. Moreover, if $x = y$ (i.e., O_a and O_b were generated at the same site), then it must be that $O_b \rightarrow O_a$ due to the FIFO property of a TCP connection. Therefore, condition $(x = y \ \textbf{and} \ T_{O_b}[y] > T_{O_a}[2])$ can never be satisfied and does not need to be checked either. Consequently, formula (6) can be simplified as:

$$O_a \parallel O_b \iff x \neq y \ \textbf{and} \ (\sum_{j=1, j\neq x}^{N} T_{O_b}[j]) > T_{O_a}[1]. \quad (7)$$

5 An example

To illustrate the operation timestamping and concurrency checking schemes, the same collaborative editing scenario in Fig. 2 is adapted to include the state vector at each site and timestamping information for each operation, as shown

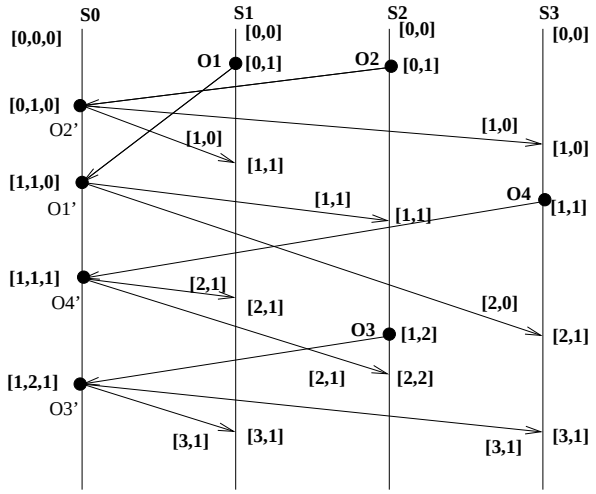

Figure 3: A scenario for illustrating the compressed state vector timestamping and concurrency checking schemes

in Fig.3. Site 0 maintains a 3-element full state vector, and the other sites maintain a 2-element compressed state vector.

First, it should be pointed out that there is a major difference between Fig. 3 and Fig. 2: the propagated operations from site 0 are in *transformed* forms (denoted by O_1', O_2', O_3', and O_4') in Fig. 3 but in *original* forms in Fig. 2. Because of this difference, operations O_1', O_2', O_3', and O_4' can be treated as if they were operations originally generated at site 0. In other words, the transformed operations at site 0 are different from their corresponding original operations (so four new operation bullets are explicitly added in Fig. 3), and their causality relationship with other operations are also different from their original operations. For example, we have $O_1 \parallel O_2$, but $O_2 \rightarrow O_1'$ because O_1' is an operation different from O_1 and the execution of O_2 happened before the *generation* of O_1' at site 0 (see Definition 1).

To explain how the compressed state vector timestamping and concurrency checking schemes can correctly capture the causality relationship among the transformed and original operations in Fig. 3, we examine the processes of handling the four original operations one by one in the order of their arrival at site 0.

Handling operation O_2

After O_2 is generated and executed at site 2 (with an empty HB_2), it is timestamped by $[0, 1]$ and propagated to site 0. Then, it is buffered in $HB_2 = [O_2]$.

When O_2 arrives at site 0 (with an empty HB_0), it is executed as-is (i.e., $O_2' = O_2$). Then, O_2' is timestamped (1) by a compressed state vector $[1, 0]$ and propagated to site 1; (2) by a compressed state vector $[1, 0]$ and propagated to site 3; and (3) by the current full state vector $[0, 1, 0]$ and buffered in $HB_0 = [O_2']$.

When O_2' arrives at site 1 (with $HB_1 = [O_1]$), it is checked against O_1 for concurrency relationship and found that $O_2' \parallel O_1$ because $(T_{O_1}[2] = 1) > (T_{O_2'}[2] = 0))$. Then, O_2'

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:19:02 UTC from IEEE Xplore. Restrictions apply.

is transformed against the concurrent operation O_1 and executed. Next, the executed O_2' is buffered in $HB_1 = [O_1, O_2']$.

When O_2' arrives at site 3 (with an empty HB_3), it is executed as-is and buffered in $HB_3 = [O_2']$.

Handling operation O_1

After O_1 is generated and executed at site 1 (with an empty HB_1), it is timestamped by $[0, 1]$ and propagated to site 0. Then, it is buffered in $HB_1 = [O_1]$.

When O_1 arrives at site 0 (with $HB_0 = [O_2']$), it is checked against O_2' for concurrency relationship and found that $O_2' \parallel O_1$ because they were generated at different sites and $(\sum_{j=1,j\neq 1}^{3} T_{O_2}[j] = 1) > (T_{O_1}[1] = 0)$. Then, O_1 is transformed against the concurrent operation O_2' to produce O_1' and executed. Next, O_1' is timestamped (1) by a compressed state vector $[1, 1]$ and propagated to site 2; (2) by a compressed state vector $[2, 0]$ and propagated to site 3; and (3) by the current full state vector $[1, 1, 0]$ and buffered in $HB_0 = [O_2', O_1']$.

When O_1' arrives at site 2 (with $HB_2 = [O_2]$), it is checked against O_2 for concurrency relationship and found $O_2 \nparallel O_1'$ because $T_{O_2}[2] = T_{O_1'}[2] = 1$. Consequently, O_1' is executed at site 2 as-is and buffered in $HB_2 = [O_2, O_1']$.

When O_1' arrives at site 3 (with $HB_3 = [O_2', O_4]$), it is checked against O_2' and O_4 for concurrency relationship and found: (1) $O_2' \nparallel O_1'$ because $T_{O_2'}[2] = T_{O_1'}[2] = 0$; and (2) $O_4 \parallel O_1'$ because $(T_{O_4}[2] = 1) > (T_{O_1'}[2] = 0)$. Then, O_1' is transformed against the concurrent operation O_4, executed, and buffered in $HB_3 = [O_2', O_4, O_1']$.

Handling operation O_4

After O_4 is generated and executed at site 3 (with $HB_3 = [O_2']$), it is timestamped by $[1, 1]$ and propagated to site 0. Then, it is buffered in $HB_4 = [O_2', O_4]$.

When O_4 arrives at site 0 (with $HB_0 = [O_2', O_1']$), it is checked against O_2' and O_1' for concurrency relationship and found: (1) $O_2' \nparallel O_4$ because $\sum_{j=1,j\neq 3}^{3} T_{O_2'}[j] = T_{O_4}[1] = 1$; and (2) $O_1' \parallel O_4$ because they were generated at different sites and $(\sum_{j=1,j\neq 3}^{3} T_{O_1'}[j] = 2) > (T_{O_4}[1] = 1)$. Then, O_4 is transformed against the concurrent operation O_1' to produce O_4' and executed. Next, O_4' is timestamped (1) by a compressed state vector $[2, 1]$ and propagated to site 1; (2) by a compressed state vector $[2, 1]$ and propagated to site 2; and (3) by the current full state vector $[1, 1, 1]$ and buffered in $HB_0 = [O_2', O_1', O_4']$.

When O_4' arrives site 1 (with $HB_1 = [O_1, O_2']$), it is checked against O_1 and O_2' for concurrency relationship and found: (1) $O_1 \nparallel O_4'$ because $T_{O_1}[2] = T_{O_4'}[2] = 1$; and (2) $O_2' \nparallel O_4'$ because $(T_{O_2'}[2] = 0) < (T_{O_4'}[2] = 1)$. Then, it is executed as-is and buffered in $HB_1 = [O_1, O_2', O_4']$.

When O_4' arrives site 2 (with $HB_2 = [O_2, O_1', O_3]$), it is checked against O_2, O_1', and O_3 for concurrency relationship and found: (1) $O_2 \nparallel O_4'$ because $T_{O_2}[2] = T_{O_4'}[2] = 1$; (2) $O_1' \nparallel O_4'$ because $T_{O_1'}[2] = T_{O_4'}[2] = 1$; and (3) $O_3 \parallel O_4'$ because they were generated at different sites and $(T_{O_3}[2] = 2) > (T_{O_4'}[2] = 1)$. Then, O_4' is transformed against the concurrent operation O_3, executed, and buffered in $HB_2 = [O_2, O_1', O_3, O_4']$.

Handling operation O_3

After O_3 is generated and executed at site 2 (with $HB_2 = [O_2, O_1']$), it is timestamped by $[1, 2]$ and propagated to site 0. Then, it is buffered in $HB_2 = [O_2, O_1', O_3]$.

When O_3 arrives at site 0 (with $HB_0 = [O_2', O_1', O_4']$), it is checked against O_2', O_1', and O_4' for concurrency relationship and found: (1) $O_2' \nparallel O_3$ because they were generated at the same site 2; (2) $O_1' \nparallel O_3$ because $\sum_{j=1,j\neq 2}^{3} T_{O_1'}[j] = T_{O_3'}[1] = 1$; and (3) $O_4' \parallel O_3$ because $(\sum_{j=1,j\neq 2}^{3} T_{O_4'}[j] = 2) > (T_{O_3}[1] = 1)$. Then, O_3 is transformed against the concurrent operation O_4' to generate $O_3^p rime$ and then executed. Next, O_3' is timestamped (1) by a compressed state vector $[3, 1]$ and propagated to site 1; (2) by a compressed state vector $[3, 1]$ and propagated to site 3; and (3) by the current full state vector $[1, 2, 1]$ and buffered in $HB_0 = [O_2', O_1', O_4', O_3']$.

When O_3' arrives site 1 (with $HB_1 = [O_1, O_2', O_4']$), it is checked against O_1, O_2', and O_4' for concurrency relationship and found: (1) $O_1 \nparallel O_3'$ because $T_{O_1}[2] = T_{O_3'}[2] = 1$; (2) $O_2' \nparallel O_3'$ because $(T_{O_2'}[2] = 0) < (T_{O_3'}[2] = 1)$; and (3) $O_4' \nparallel O_3'$ because $T_{O_4'}[2] = T_{O_3'}[2] = 1$. Then, it is executed as-is and buffered in $HB_1 = [O_1, O_2', O_4', O_3']$.

When O_3' arrives site 3 (with $HB_3 = [O_2', O_4, O_1']$), it is checked against O_2', O_4, and O_1' for concurrency relationship and found: (1) $O_2' \nparallel O_3'$ because $(T_{O_2'}[2] = 0) < (T_{O_3'}[2] = 1)$; (2) $O_4 \nparallel O_3'$ because $T_{O_4}[2] = T_{O_3'}[2] = 1$; and (3) $O_1' \nparallel O_3'$ because $(T_{O_1'}[2] = 0) < (T_{O_3'}[2] = 1)$. Then, it is executed as-is and buffered in $HB_3 = [O_2', O_4, O_1', O_3']$.

It can be verified that the concurrency relationship among all operations detected by compressed state vector timestamping in this example indeed correctly captures the causality relationship among all operations as defined by Definition 1.

6 Conclusions

We have proposed and discussed a novel approach to compressing a vector clock timestamp into a constant size of 2 for a system with an arbitrary number N of communicating processes. This approach was motivated by our work in real-time group editors, and has taken advantage of a novel operational transformation technique to transform the N-dimension causality relationship into a 2-dimension causality relationship among operations. We have shown how compressed vector clocks can be used as an effective and efficient means for operation timestamping and concurrency detection in group editors. The proposed compressed state vector technique has been implemented in a web-based real-time group editing demonstrator (http://reduce.qpsf.edu.au),

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:19:02 UTC from IEEE Xplore. Restrictions apply.

which allows an arbitrary number of users to participate a collaborative editing session.

To the best of our knowledge, our vector clock compressing technique is unique in compressing the size (from N to 2) of vector clocks by transforming the dimension (from N to 2) of causality relationship among operations. Our approach has the advantage of substantially reducing the vector-caused communication overhead to a minimum of two integers, rather than being linear in N as in early compressing techniques [9, 13]. In addition, all communicating processes in our system, except the notifier, need to maintain a single vector of 2 elements only, rather than having to maintain three full vectors of N elements by every process as in early compressing techniques [9, 13].

It is worth pointing out explicitly that the key to achieving vector clock compression in our scheme is the novel operational transformation technique, which enables the center notifier to convert the N-dimensional causality relationships among operations into 2-dimensional relationships. If the notifier propagates operations as-is (i.e., without transformation), the causality relationships among these operations would still remain N-dimensional and have to be timestamped by N-element vector clocks.

Although our compression scheme was designed in the context of Internet-based real-time group editing systems, the basic ideas and techniques in this scheme are potentially applicable to other distributed systems which support concurrent updates on replicated data objects, such as replicated database systems, replicated file systems, etc. The applicability of our approach depends on the availability of an application-dependent technique (like operational transformation) which is able to transform messages in such a way that the N-dimension causality relationship among messages can be converted into 2-dimension causality relationship. We hope the work presented in this paper could provide some inspirations for researchers seeking innovative solutions for efficient implementation of vector clocks in advanced distributed computing systems and applications.

Acknowledgments

The authors wish to thank the anonymous referees for their very constructive comments and suggestions. The work reported in this paper has been partially supported by an ARC (Australia Research Council) Large Grant (No: A00000711). Part of this work was carried out during the first author's sabbatical leave at the School of Computer Engineering, Nanyang Technological University, Singapore.

References

[1] K. Birman, A. Schiper, and P. Stephenson: "Lightweight causal and atomic group multicast," *ACM Trans. on Comp. Sys.* 9(3) (August), 1991, pp.272–314.

[2] W. Cai, K. Zhang, S.J. Turner, and C. Sun: "Interlock Avoidance in Transparent and Dynamic Parallel Program Instrumention Using Logical Clocks," *Parallel Computing*, (25)5, pp.569-591, North-Holland, Elsevier Science Publishers B.V., May, 1999.

[3] W. Cai, Bu-Sung Lee, and Junlan Zhou: "Causal order delivery in multicast environment: an improved algorithm," Journal of Parallel and Distributed Computing, Vol.62, No.1, Jan 2002.

[4] B. Charron-Bost: "Concerning the size of logical clocks in distributed systems," *Information Processing Letters*, 39, pp.11-16, July 1991.

[5] C. A. Ellis and S. J. Gibbs: "Concurrency control in groupware systems," In *Proc. of ACM Conference on Management of Data*, pp.399-407, 1989.

[6] C. Fidge: "Timestamps in message-passing systems that preserve the partial ordering," In *Proceedings of the 11th Australian Computer Science Conference* (1988), pp.56-66.

[7] J. Fowler and W. Zwaenepoel: "Causal distributed breakpoints," *Proc. of 10th Int. Conference on Distributed Computing Systems*, Paris, France, pp.134-141, May 1990.

[8] L. Lamport: "Time, clocks, and the ordering of events in a distributed system," *CACM 21(7)*, pp.558-565, July 1978.

[9] S. Meldal, S. Sankar, and J. Vera: "Exploring locality in maintaining potential causality," *Proc. of 10th ACM Symposium on Principles of Distributed Computing*, Montreal, Canada, pp.231-239, 1991.

[10] M. Raynal and M. Singhal:"Logical time: capturing causality in distributed systems," *IEEE Computer*, pp.49-56, Feb. 1996.

[11] R. Ronngren and M. Liljenstam: "On event ordering in parallel discrete event simulation," In *Proc. of the 13th Workshop on Parallel and Distributed Simulation (PADS'99)*, pp.38-45, 1999.

[12] R.Schwarz and F. Mattern: "Detecting causal relationships in distributed computations: in search of the holy grail," *Distributed Computing*, 7(3):149-174, 1994.

[13] M. Singhal and A. Kshemkalyani: "An efficient implementation of vector clocks," *Information Processing letters*, 43, pp. 47-52, Aug. 1992.

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

[15] C. Sun and C. A. Ellis: "Operational transformation in real-time group editors: issues, algorithms, and achievements," In *Proc. of ACM Conference on Computer-Supported Cooperative Work*, pp.59-68, Seattle, USA, Nov.14-18, 1998.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:19:02 UTC from IEEE Xplore. Restrictions apply.