

# DCV: A Causality Detection Approach for Large-scale Dynamic Collaboration Environments\*

Ning Gu, Qiwei Zhang, Jiangming Yang and Wei Ye  
Department of Computing and Information Technology  
Fudan University  
No.220 Handan Road, Shanghai 200433, P.R.China  
{ninggu, qiweizhang, yangjiangming, weiye}@fudan.edu.cn

## ABSTRACT

Recent studies have indicated the significance of supporting real-time group editing in "Wiki" applications, whose collaboration environments have their dynamic and large-scale nature. Correct capture of causal relationships between operations from different users is crucial in order to preserve consistency of object copies. This challenge was resolved by employing vector logical clock. But since its size is equal to the number of cooperating sites, it has low efficiency when dealing with a collaborative environment involving a large number of participants. In this paper, we propose a direct causal vector (DCV) approach for solving causality detection issues in real-time group editors. DCV timestamp does not record the causality information that can be deduced from the transitivity of causal relation. As a result, it can automatically reduce its own size when people leave the collaboration session and always keep small. We prove that DCV approach is well fit for capturing causality in wiki like large-scale dynamic collaboration environments.

## Categories and Subject Descriptors

C.2.4 [Computer-Communication Networks]: Distributed applications; K.4.3 [Computers and Society]: Computer supported collaborative work

## General Terms

Algorithms, Design

## Keywords

CSCW, Concurrency Control, Groupware, Group Editors, Logical Clock, Direct Causal Vector

\*The work is supported by National Natural Science Foundation of China (NSFC) under Grant No.90612008, National Grand Fundamental Research 973 Program of China under Grant No.2005CB321905 and Shanghai Science and Technology Committee Key Fundamental Research Project under Grant No.05JC14006.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GROUP'07, November 4-7, 2007, Sanibel Island, Florida, USA.  
Copyright 2007 ACM 978-1-59593-845-9/07/0011 ...\$5.00.

## 1. INTRODUCTION

Real-time group editors enable a group of users to view and edit the same document (e.g., text, graphic and multimedia document) simultaneously from geographically dispersed sites connected by communication networks such as the Internet[2, 7]. This category of groupwares have some unique characteristics which make them distinct from other kinds of collaborative work[2, 4, 7, 8, 22]: 1. real-time: the response to local user actions must be quick and the latency for reflecting remote user actions must be low; 2. unconstrained: cooperating users are able to freely edit any part of the document at any time, synchronously or not. 3. distributed: cooperating users may operate different machines connected by different communication networks with non-deterministic latency.

A replicated approach has been proposed to meet the requirements for good responsiveness and unconstrained collaboration. Documents are replicated at each cooperating site, and user operations are first executed on their local replicas immediately, and then propagated to all other cooperating sites for group awareness. To ensure that the document replicas at each cooperating sites can always exactly reflect the initial intentions of each cooperating users, and can eventually arrive at the same consistent status at the end of collaborative session, whenever to synchronize a remote operation, it must take into account the impact of all the concurrent operations executed previously. Thus, precise capture of causality is crucial in real-time group editors[2, 7, 8].

Vector logical clock is currently the most widely used causality detection technique[2, 7, 9, 17, 18, 19]. However, it has only been proved to work well in small collaboration environment (less than ten participants). Our recent studies have indicated the significance of supporting real-time group editing in "Wiki" applications, whose collaboration environments have their dynamic and large-scale nature. Vector logical clock is not fit for such collaboration environments. Is there any other causality detection solution? This is the focus of this paper.

### 1.1 Real-time Wiki Editing and Large-scale Dynamic Collaboration Environment

The Internet has fostered an unconventional and powerful style of collaboration: "wiki" web sites. A *wiki* is such a website — every visitor is allowed and able to add new pages to it, remove existing pages, or otherwise edit and change the content of existing pages, using just web browsers[25]. By working on the same wiki website, geographically dispersed

users can easily collaborate and share knowledge with each other. Wiki web sites, as a novel and convenient collaboration medium, is becoming more and more popular, which could be evidenced by an increasing number of wiki applications, such as Wikipedia[26], WikiNews[24], WikiTravel[28] and StrategyWiki[23].

Existing wikis only support asynchronous collaboration. People collaborate with each other in a Copy-Modify-Merge manner: each wikipedian checks out a separate working copy of wiki pages from the the wiki server; then modifies her/his working copy independently; and finally merges her/his working copy with those from other wikipedians[23, 24, 26, 28]. Our recent study means to, by adapting existing single-user wiki page editors to full-featured real-time group editors, support wiki users synchronous collaboration.

Supporting both synchronous and asynchronous collaboration allows more flexible collaboration among users. Wiki users can collaborate with each other more closely and more effectively. Consider the following scenario: a wiki user, named Bob, poor at language expression, just drafts his ideas on the wiki page; at the same time another wiki user help to revise his expression; if there is misunderstanding, Bob will immediately notice and correct it. It could save Bob a lot of time without carefully choosing and organizing his words.

Another notable benefit is that it could help to eliminate editing conflicts. In existing wikis, we have more than once experienced that we made a large number of changes to a paragraph of text, only to have them all silently discarded when someone else, editing the same paragraph at the same time, saved their modifications after me. Although this does not happen very often, they are really frustrating. As has been mentioned before, wiki systems are gaining more and more populations. As the wiki systems grow, so do the opportunities for conflict. How to help reducing the coordination costs? It has become a problem faced by many wiki designers and researchers[1]. An unique feature of real-time group editors is real-time workspace awareness. It could help people detect editing conflicts as early as possible. Historical studies have indicated that when concurrently working on the same object, if people are able to see others in real time, it is less likely that their actions seriously conflict with each other[3].

The collaborative environments in wikis are typically large-scale and dynamic collaboration environments. In wiki applications, each wiki page represents a collaboration session. People viewing or editing one wiki page all are participants of corresponding wiki collaboration session. The number of participants of a wiki collaboration session could be very large (i.e. hundreds, thousands or more). Take Wikipedia as an example. Many of its wiki pages have over thousands of hundreds of visitors a day[27]. Another feature of wiki collaboration environments is dynamic. In wiki applications like Wikipedia, every minute there are hundreds of users join and leave[27]. People usually leave a wiki collaboration session (wiki page) by simply closing the web browser or navigating to another web page. There is not any notification.

## 1.2 Existing Approaches: The Problems

Most existing real-time group editors employ vector logical clock to help detect causality[2, 7, 9, 17, 18, 19]. In these implementations, a fixed sized state vector is attached

to each operation  $o$ , named vector logical clock timestamp, in which each item corresponds to a collaboration participant, and records the number of operations generated by that participant that are causally preceding  $o$ . By comparing their vector logical clock timestamps, causal relationship between any two operations can be easily determined.

This approach works well when the collaboration group is small. However, the size of vector logical clock linearly depends on the number of participants. If there are hundreds, thousands or even more participants, problem occurs. The timestamp attached to each operation will become huge in size, even much larger than the operation itself, which could add an intolerable cost to network transmission, local storage as well as causality detection complexity, and as a result, seriously affect the performance of the groupware system. Moreover, many large-scale collaboration environments (i.e. Wikipedia) are open. They tend to have a nondeterministic number of participants. In such collaboration environments, it is not practical to timestamp operations with a fixed size state vectors.

Some previous work tries to solve this problem by[11]: 1. redesigning the vector logical clock, using associative vectors indexed by participant identifier, thus allowing the system to dynamically add new timestamp items or discard old timestamp items during the collaboration session; and 2. allocating timestamp items dynamically at run-time, creating vector items just for those participants who have written, instead of pre-allocating a timestamp item for all the potential participants; and 3. if some participants midway leave the collaboration session, watching their corresponding timestamp items and removing them once they have become insignificant.

In these above approaches, since timestamp items are not pre-allocated at the beginning of the collaboration session, opening large-scale collaboration environments should no longer be a problem. Furthermore, if the collaboration environment is not active, most participants just read instead of writing, the vector logical clock timestamp of each operation will never grow too large. It seems that these approaches can help to significantly ease the problems about vector logical clock in large-scale collaboration environments. Nevertheless, the trouble is that, firstly, it is difficult to determine whether a participant has really left the collaboration session or not in a large-scale collaboration environment. We still take wiki as an example. As mentioned before, people in wiki collaboration sessions often leave by simply closing the browser window or navigating to another page, without sending any notification to wiki server. Thus, we can by not direct means determine whether a participant has left the collaboration session or not. It is also not safe to assume that any participants who are not active for a pre-determined time has already left the collaboration session. People collaborate with each other over Internet. The network is unstable. Sometimes we do not receive any message from a participant for a long time, just because the network connection between he/she and other participants is temporarily failed. Even if we can make sure that a particular participant has left the collaboration session already, it is not always safe to remove her/his corresponding timestamp item. A timestamp item could be safely reclaimed only after the operations that were generated by the corresponding participant have all been synchronized at all the cooperating sites. In an unreliable and highly dynamic collaboration

environment like wiki, it is usually difficult to determine the exact list of participants that are currently present in the collaboration session and thus is also difficult to determine whether a timestamp item has become insignificant or not.

In the past decade, there have been several different techniques about vector logical clock compression[6, 11, 14, 16, 20, 21]. But as to our knowledge, none of them can really solve all its efficiency issues in large-scale collaboration environments, without imposing any assumption on network reliability and negatively affecting the users' normal collaboration work.

Noticing the limitation of vector logical clock, some existing real-time group editing systems design their own causality detection solutions[10, 13, 19]. Their main ideas are quite similar. In these systems, before propagating an local operation, cooperating sites pre-transform it against some concurrent operations generated from other cooperating sites to make them appear as sequential operations. By carefully designing the transforming and propagating rules, causality detecting issues in these systems are greatly simplified, and thus bring the possibility for much simpler causality detection solutions. The common drawback of such approaches is that, during the pre-transforming, intention conflict among different participants may be improperly hidden, which may incur much intolerable confusion. Furthermore, as to our knowledge, existing systems of this category either suffer from single-point failure, or require extremely reliable and stable collaboration network. None of them is well fit for wiki like wide-area collaboration environments.

### 1.3 Overview

In this paper, we develop a new causality detecting approach, named Direct Causal Vector (DCV). This is a well scalable approach, which works well in many large-scale dynamic collaboration environments, including most existing wiki environments.

The rest of this paper is organized as follows: Section 2 introduces some background concepts and makes clear the causality detection issues needed to be solved in realtime group editor systems. Section 3 introduces the concept of direct causal vector, which is our new timestamp design. Section 4 proposes a solution to all the issues proposed in section 2, based on our new timestamp design. Section 5 makes some discussion on our new approach and section 6 compares it with related works. Section 7 summarizes the contributions of this paper.

## 2. BACKGROUND

In this section, we will introduce some background concepts and articulate the causality detection issues needed to be solved in realtime group editor systems.

### 2.1 Causal Relation

Following Lamport[15], in this paper, we define a causal ordering relation on operations in terms of their generation and execution sequences.

*Definition 1. [Causal relation "→"]* Given two operations  $o_a$  and  $o_b$ , generated at site  $i$  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  $o_x \rightarrow o_b$ .

The operation  $o_a$  is said to causally precede  $o_b$  iff  $o_a \rightarrow o_b$ . Operations that are not causally related are said to be concurrent, denoted as  $o_a \parallel o_b$ . More accurately  $o_a$  and  $o_b$  are concurrent iff neither  $o_a \rightarrow o_b$  nor  $o_b \rightarrow o_a$ .

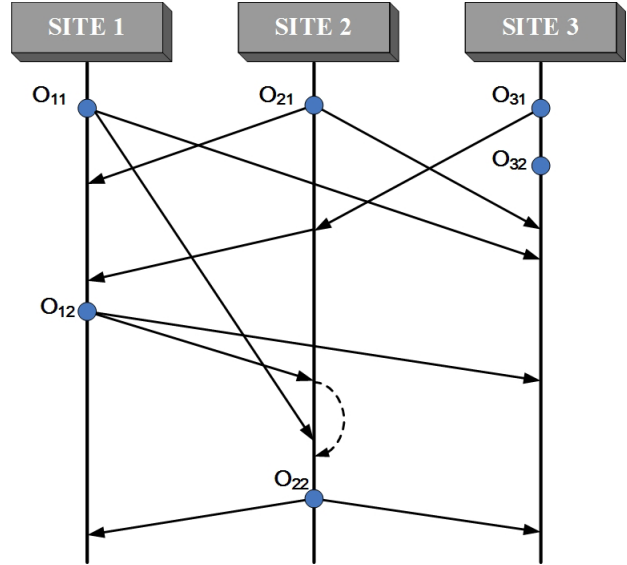


Figure 1: A scenario of a real-time cooperative editing session.

As illustrated in Figure 1.  $o_{11}$ ,  $o_{21}$  and  $o_{31}$  are concurrent with each other.  $o_{12}$  is causally dependent on three operations  $o_{11}$ ,  $o_{21}$  and  $o_{31}$ .  $o_{22}$  is causally dependent on four operations  $o_{11}$ ,  $o_{21}$ ,  $o_{31}$  and  $o_{12}$ .  $o_{32}$  is only causally dependent on  $o_{31}$

### 2.2 Causality Detection Issues

To achieve high responsiveness and high concurrency, a replicated architecture has been introduced and widely applied in current realtime group editor systems. In this architecture, a document replica is maintained at each cooperating site. Collaborators are allow to modify their own document replicas at any time. After their local executions, operations are broadcasted to all the collaborators for synchronization. For high usability, it is required that a real-time group editor with replicated architecture always satisfy the following properties[2, 5, 7]:

1. Convergence: When the same set of operations have been executed at all sites, all copies of the shared document are identical.
2. Causality preservation: For any pair of operations  $o_a$  and  $o_b$ , if  $o_a \rightarrow o_b$ , then  $o_a$  is executed before  $o_b$  at all sites.
3. Intention preservation: For any operation  $o$ , the effects of executing  $o$  at all sites are the same as the intention of  $o$ , and the effect of executing  $o$  does not change the effects of independent operations.

Thus, the following causality detection issues surface.

*Problem 1.* Given an unexecuted remote operation, how to determine whether all the operations causally preceding it have already been executed?

To achieve causality preservation, we must ensure that an operation is not executed on a remote site until all the operations on which it causally depends have already been executed. However, due to the nondeterministic communication latency, remote operations may arrive out of their natural causal order. Thus, we must check its causality condition every time before executing a remote operation.

*Problem 2.* Given an unexecuted remote operation that is causally ready, how to separate all the history operations that are concurrent to it from the operation history?

An operation history is a set of history operations that satisfies (1) it contains all the history operations that have the possibility of being concurrent with the future incoming remote operations and (2) if it contains an operation, all the history operations causally depend on this operation are also included. In real-time group editor systems, an operation history could be found maintained at each cooperating site [2, 7, 9, 17, 18, 19].

To achieve intention preservation, whenever executing a remote operation, the impact of concurrent operations in the operation history must all be taken into account.

*Problem 3.* How to determine the causal relationship between two arbitrary operations in the operation history?

It is not a trivial matter to find the real intention of a remote operation. It is also required by some concurrent control algorithms that causal relationship between two history operations be determined correctly and efficiently [2, 8, 18].

### 3. DIRECT CAUSAL VECTOR

Causal relation is transitive, that is, if  $o_a \rightarrow o_b$  and  $o_b \rightarrow o_c$ , then  $o_a \rightarrow o_c$ . Some causal relationships can be deduced from other causal relationships by transitivity, while others can not. By differentiating these two kinds of causal relationships, we come up with the concept of *direct causal relation*. Direct causal relation represents those causal relationships that can not be deduced from other causal relationships by transitivity.

*Definition 2. [Direct causal relation "⇒"]* Given two operations  $o_a$  and  $o_b$ ,  $o_a$  causally precede  $o_b$  directly, denoted as  $o_a \Rightarrow o_b$ , iff: (1)  $o_a \rightarrow o_b$ , and (2) there exists no operation  $o_x$  satisfying  $o_a \rightarrow o_x$  and  $o_x \rightarrow o_b$ .

The concept of direct causal relation just excludes those causal relationships that could be deduced from other causal relationships by transitivity, thus there exists the following theorem.

*Definition 3. [Uninterrupted Operation Set]* In this paper, we refer to an operation set as uninterrupted iff: given two causally related operations  $o_a$  and  $o_b$  in the operation set (suppose  $o_a \rightarrow o_b$ ), there exists no operation  $o$  such that  $o_a \rightarrow o$  and  $o \rightarrow o_b$  but  $o$  is not contained in the operation set.

**THEOREM 1.** *Given an uninterrupted operation set  $S$  and two operations  $o_a, o_b$  in it, then  $o_a \rightarrow o_b$ , iff (1)  $o_a \Rightarrow o_b$ , or (2) there exists a operation sequence  $o_1, o_2, \dots, o_n$  in  $S$ , such that  $(o_a \Rightarrow o_1) \wedge (o_1 \Rightarrow o_2) \wedge \dots \wedge (o_{n-1} \Rightarrow o_n) \wedge (o_n \Rightarrow o_b)$ .*

*Definition 4. [Direct causal vector(DCV)]* Given an operation  $o$ , suppose there exist only  $k$  operations that are causally preceding it directly, enumerated as  $o_1, o_2, \dots, o_k$ . Here,  $o_1$  is the  $n_1$ th operation generated at cooperating site  $s_1$ ;  $o_2$  is the  $n_2$ th operation generated at cooperating site  $s_2$ ; ...;  $o_k$  is the  $n_k$ th operation generated at cooperating site  $s_k$ . The direct causal vector of  $o$ , denoted as  $DCV(o)$ , is defined as  $DCV(o) = [(s_1, n_1), \dots, (s_k, n_k)]$ .

Notice that each item in the direct causal vector  $DCV(o)$  represents an operation that is causally preceding  $o$  directly. Each operation that is causally preceding  $o$  directly must have a corresponding item in the direct causal vector  $DCV(o)$ . It must hold that  $o_x \Rightarrow o$  iff  $o_x \in DCV(o)$ . Thus, to determine the direct causal relationship between  $o_1$  and  $o_2$ , we just need to check whether  $o_1$  exists in  $DCV(o_2)$  and whether  $o_2$  exists in  $DCV(o_1)$ .

To illustrate the concepts of *direct causal relation* and *direct causal vector*, refer to the scenario in Figure 1 again,  $o_{11}, o_{21}$  and  $o_{31}$  have no operation causally preceding them directly;  $o_{12}$  causally depends on  $o_{11}, o_{21}$  and  $o_{31}$  directly;  $o_{22}$  causally depends on  $o_{12}$  directly.  $o_{32}$  causally depends on  $o_{31}$  directly. The direct causal vector of  $o_{11}, o_{21}, o_{31}, o_{12}, o_{22}$  and  $o_{32}$  is  $[], [], [], [(1, 1), (2, 1), (3, 1)], [(1, 2)]$  and  $[(1, 3)]$  respectively.

Following are some interesting properties of direct causal relationship, related with causality detection in real-time group editor systems.

*Property 1.* *Given a remote operation  $o$ , suppose all the operations executed before respect their causal order.  $o$  is causally ready iff all the operations causally precede  $o$  directly have already been executed.*

To prove this property, consider an arbitrary operation  $o_t$  that is causally preceding  $o$  but not directly. According to theorem 1, there must exist another operation  $o_x$  satisfying  $(o_t \rightarrow o_x) \wedge (o_x \Rightarrow o)$ . According to the assumption,  $o_x$  has already been executed, and it was executed after it had become causally ready. Thus  $o_t$  must have been executed too.

This property indicates that to decide whether an unexecuted remote operation is causally ready, we just need to check the operations causally preceding it directly.

*Property 2.* *Assume  $o$  is an unexecuted remote operation that is causally ready already.  $HB$  is the operation history. The causal relationship between  $o$  and an arbitrary history operation  $o'$  in  $HB$  can be determined correctly just by tracing the direct causal relationships over  $HB \cup o$ . So is the causal relationship between two arbitrary causal operations in  $HB$ .*

Notice that  $HB \cup o$  is an uninterrupted operation set, so this property can be deduced directly from the theorem 1.

These two properties indicate that, it is possible to solve the three causality detection issues in real-time group editors just through the direct causal vector timestamp of each operations.

### 4. THE CAUSALITY DETECTION SCHEME

In this section, we will look into the causal detection issues mentioned in section 2 and give a direct causal vector timestamp based solution for each of these issues.

## 4.1 Causality preservation

As has been pointed out in the previous section, to solve the first causality detection issue, that is, to decide whether an unexecuted remote operation is causally ready, we only need to check whether all the operations causally preceding it directly have already been executed.

As is done in many previous works, we use state vector. Each site  $s$  maintains a state vector  $SV_s$ . For each site  $t$  in the collaboration session, there is a corresponding component in  $SV_s$ , denoted as  $SV_s[t]$ , which holds a value corresponding to the number of operations that were generated at site  $t$  and have already been synchronized on site  $s$ . A remote operation  $o_r$  received on site  $s$  is thought to be causally ready, if and only if it holds for each item  $(s_i, n_i)$  in  $DCV(o_r)$  that  $SV_s[s_i] \geq n_i$ .

For illustration, refer to the scenario in Figure 1 again. At the time when  $o_{12}$ , whose direct causal vector timestamp is  $[(1, 1), (2, 2), (3, 3)]$ , is received at site 2, only two operations, named  $o_{21}$  and  $o_{31}$ , have been executed at site 2, that is, the state vector of site 2 is  $\{SV_2[1] = 0; SV_2[2] = 1; SV_2[3] = 1\}$ . It is not satisfied that  $SV_2[1] \geq 1$ , so  $o_{12}$  is not causally ready and its synchronization must be delayed. The synchronization of  $o_{12}$  is delayed until  $o_{11}$  has been received and synchronized at site 2. At that time, the state vector at site 2 has changed to  $\{SV_2[1] = 1; SV_2[2] = 1; SV_2[3] = 1\}$ . It satisfies that for each item  $(s_i, n_i)$  in  $DCV(o_{12})$ ,  $SV_2[s_i] \geq n_i$ . Thus, we could make sure that  $o_{12}$  has become causally ready for synchronization.

## 4.2 Concurrent Separation

In this subsection, we will focus on the second causality detection issue, that is, given an unexecuted remote operation  $o_r$  that is causally ready, separating from the operation history all the operations that are concurrent with it.

Our basic idea is to check the history operations (the operations in operation history) in the reverse order of their causal order. And we do not check a history operation until all the history operations that are causally dependent on it have been proved to be concurrent with  $o_r$ .

Not all the history operations will be checked during the process. But if a history operation is ignored, there must exist at least one of its direct causal successor in the operation history that could be proved to be causally preceding  $o_r$ . Such operation can not be concurrent with  $o_r$ , thus ensuring the correctness of our approach.

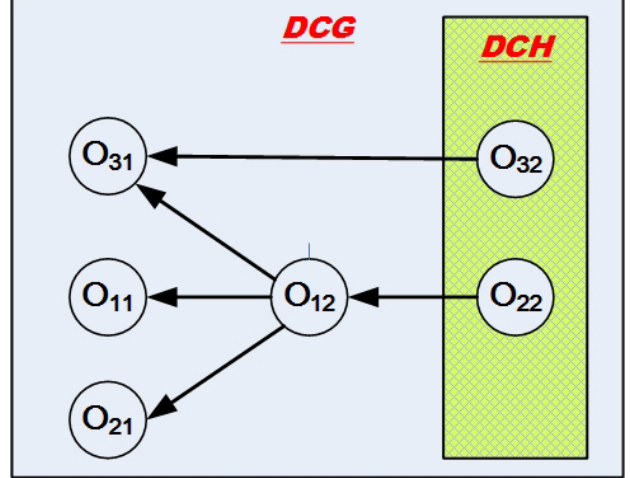
Concerning a history operation  $o_h$ , whose direct causal successors have all been proved to be concurrent with  $o_r$ . Due to Theorem 1, it must hold that  $o_h$  is either concurrent with  $o_r$  or directly causally preceding  $o_r$ . Thus, to decide whether  $o_h$  and  $o_r$  are concurrent with each other, we just need to check their direct causal relationship. As has been pointed out in the previous section, the direct causal relationship between  $o_h$  and  $o_r$  can be determined directly by checking the direct causal vector timestamp of  $o_r$ , thus ensuring the efficiency of our approach.

We propose two data structures to facilitate the concurrent separating process: direct causal history and direct causal graph.

Direct causal history (DCH) is a subset of operation history. It contains those operations that have no causal successor in the operation history.

Direct causal graph (DCG) is a graph of history operation. For each operation in the operation history, there is a corre-

sponding node in the direct causal graph. Given two history operations  $o_a$  and  $o_b$ , there is a link from  $o_a$  to  $o_b$  in the direct causal graph iff  $o_b \rightarrow o_a$ . In the following discussion,  $LT(o)$  will be used to denote the set of history operations that an operation  $o$  links to in the direct causal graph, and  $LF(o)$  will be used to denote the number of history operations that have a link to an operation  $o$  in the direct causal graph.



**Figure 2: Illustration of the new concurrent separation method.**

To illustrate, consider the scenario in Figure 1. At site 3, after the execution of  $o_{22}$ , there should be six operations in its operation history:  $o_{11}$ ,  $o_{21}$ ,  $o_{31}$ ,  $o_{12}$ ,  $o_{22}$  and  $o_{32}$ . The corresponding status of DCH and DCG is displayed in Figure 2.

None of the operations  $o_{11}$ ,  $o_{21}$ ,  $o_{31}$ ,  $o_{12}$  and  $o_{22}$  is causally dependent on  $o_{32}$ . Thus,  $o_{32} \in DCH$ . Similarly,  $o_{22} \in DCH$ . Other history operations all have at least one causal successor in the operation history,  $o_{11}$  as an example, the operation  $o_{12}$  contained in the operation history is one of its causal successors, so none of them belongs to DCH.

In DCG, there are six operation nodes, corresponding to the six operations in the operation history. There are five directed edges in the graph, each corresponding to a direct causal relationship over the six history operations, i.e. there is a link from  $o_{32}$  to  $o_{31}$  in the graph because  $o_{31} \rightarrow o_{32}$ , there is a link from  $o_{22}$  to  $o_{12}$  in the graph because  $o_{12} \rightarrow o_{22}$ , etc. Furthermore, according to the graph, we have  $LF(o_{32}) = LF(o_{22}) = 0$ ,  $LF(o_{12}) = LF(o_{11}) = LF(o_{21}) = 1$  and  $LF(o_{31}) = 2$ .

Following is the detail algorithm.

The *ConcurrentSeparation* procedure 1 takes a causally ready unexecuted operation  $o_r$  as input and returns an operation set  $set_{o_r}^c$  that contains all the history operations that are concurrent with  $o_r$ .

The operation set *candidates* is used to, during the procedure, temporarily keep the newly found history operations that are proved to have no direct causal successors in the operation history causally preceding  $o_r$ . It is initialized with DCH (line 2), because (1) none of the operations in DCH have causal successors in the operation history, let alone having any causal successors in the operation history causally preceding  $o_r$ , and (2) the operations in DCH are the only

history operations that could be proved to have no direct causal successors in the operation history causally preceding  $o_r$ , at the beginning of the procedure. Based on the above analysis, to select out all the history operations concurrent with  $o_r$ , we begin by checking these operations. The map structure  $map$  is used to record the number of history operations causally dependent on it directly which have been proved to be concurrent with  $o_r$  for each history operation.

---

**Algorithm 1**  $Concurrent\_Separation(o_r)$ :  $set_{o_r}^c$

---

```

1:  $set_{o_r}^c \leftarrow \phi$ 
2:  $candidates \leftarrow DCH$ 
3:  $map \leftarrow \phi$ 
4: add a node to DCG, representing operation  $o_r$ 
5: add  $o_r$  to DCH
6: repeat
7:    $examinee \leftarrow candidates$ 
8:    $candidates \leftarrow \phi$ 
9:   for all  $o_h$  in  $examinee$  do
10:    if  $o_h \rightrightarrows o_r$  then
11:      if  $o_h \in DCH$  then
12:        remove  $o_h$  from DCH
13:      end if
14:      add to DCG a link from  $o_r$  to  $o_h$ 
15:    else
16:      add  $o_h$  to  $set_{o_r}^c$ 
17:      for all  $o$  in  $LT(o_h)$  do
18:        if  $map[o]$  is undefined then
19:           $map[o] \leftarrow 1$ 
20:        else
21:           $map[o] \leftarrow map[o] + 1$ 
22:        end if
23:        if  $map[o] = LF(o)$  then
24:          add  $o$  to  $candidates$ 
25:        end if
26:      end for
27:    end if
28:  end for
29: until  $candidates$  is empty
30: return  $set_{o_r}^c$ 

```

---

The  $Concurrent\_Separation$  procedure simply repeats the checking process from line 7 to line 28 until all the concurrent history operations have been found.

In each round of checking, the  $Concurrent\_Separation$  procedure check whether it is causally dependent on  $o_r$  directly for each history operation in  $candidates$ . If not, according to the above analysis, the history operation currently checked must be concurrent with  $o_r$ , so we add it into  $set_{o_r}^c$  (line 16), and update  $map$  for each of the history operations it causally depends on directly (line 17-26). At the same time, we check whether some more operations could be proved to have no direct causal successors in the operation history causally preceding  $o_r$ . If so, we will add such operations into  $candidates$  and prepare them for a next round of checking (line 23-25). According to the definition of  $map$ , it is easy to see that if  $map[o] = LF(o)$ , it must hold that all the history operations which causally depends on  $o$  directly have been proved to be concurrent with  $o$ .

During the execution, the procedure also finish the adjustment of data structures DCH and DCG. Initially, a new node is created in DCG and DCH for  $o_r$  (line 4-5). For each

operation causally preceding  $o_r$  directly, the procedure ensures its removal from DCH and adding to DCG a link from  $o_r$  to it (line 11-14). Notice that if a history operation is causally preceding the causally ready unexecuted  $o_r$  directly, it must have no direct causal successor in the operation history causally preceding  $o_r$ . All such operations are checked and handled during the execution of  $Concurrent\_Separation$ , and thus ensures the integrity of such adjustment.

To illustrate the  $Concurrent\_Separation$  procedure, still consider the scenario in Figure 1. Suppose a new remote operation  $o_{41}$ , whose direct causal vector is  $[(1, 1), (3, 2)]$ , is received at site 3 shortly after the execution of  $o_{22}$  on that site. Obviously,  $o_{41}$  is causally ready.

At the beginning of  $Concurrent\_Separation(o_{41})$ , the operation set  $candidates$  is initialized with the two history operations in DCH  $o_{32}$  and  $o_{22}$ .

In the first round of checking,  $o_{22}$  is proved to be concurrent with  $o_{41}$ . It is put into  $set_{o_{41}}^c$ . At the same time, the value of  $map[o_{12}]$  is adjusted to 1. Noticing that the value of  $map[o_{12}]$  becomes equal to  $LF(o_{12})$ , so  $o_{12}$  is pushed into the operation set  $candidates$ .  $o_{32}$  is proved to be causally preceding  $o_{41}$ . It is removed from DCH and a link is added into DCG from  $o_{41}$  to  $o_{32}$ . During the first round of checking, only one more history operations  $o_{12}$  is proved to have no direct causal successor in the operation history causally preceding  $o_{41}$ , so at the end of this round of checking, the status of  $candidates$  becomes  $\{o_{12}\}$ .

In the second round of checking,  $o_{12}$  is proved to be concurrent with  $o_{41}$ . The status of variable  $map$  is updated to  $\{map[o_{12}] = 1; map[o_{31}] = 1; map[o_{11}] = 1; map[o_{21}] = 1\}$ .  $LF(o_{11}) = 1$  and  $LF(o_{21}) = 1$ . Two more operations  $o_{11}$  and  $o_{21}$  are proved to have no direct causal successor in the operation history causally preceding  $o_{41}$ . They are added into the operation set  $candidates$ . At the end of the second round of checking, the status of  $candidates$  is  $\{o_{11}, o_{21}\}$ .

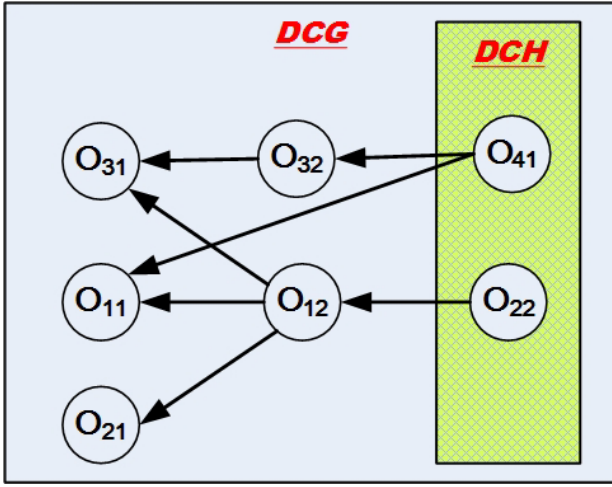
In the third round,  $o_{11}$  is proved to be causally preceding  $o_{41}$  directly, and  $o_{21}$  is proved to be concurrent with  $o_{41}$ . A link from  $o_{41}$  to  $o_{11}$  is added into DCG. No more history operation is found that could be added into  $candidates$ . Thus the status of  $candidate$  becomes empty at the end of the third round of checking.

The execution of  $Concurrent\_Separation(o_{41})$  is completed.  $set_{o_{41}}^c = \{o_{22}, o_{12}, o_{21}\}$  is returned as the result. As a side-effect of  $Concurrent\_Separation(o_{41})$ , the status of DCH and DCG is adjusted as illustrated in Figure 3.

By and large, we have described our concurrent separation algorithm in full detail. Notice that whenever a new local operation is generated, the operations in DCH exactly contains the operations it causally depends on directly. Thus, we can build direct casual vector based timestamp for a newly generated local operation directly from the content of DCH. Furthermore, after each execution of a local operation  $o_i$ , the data structures DCH and DCG must be immediately adjusted. A new node should be added into DCG corresponding to operation  $o_i$ . A link from  $o_i$  to each operations currently in DCH operation set should be added into DCG. And finally the status of DCH operation set should be adjusted to  $\{o_i\}$ .

### 4.3 Causality Cache

In this subsection, we will focus on the third causality detection issue: determining the causal relationship between two arbitrary operations in the operation history.



**Figure 3: Illustration of the new concurrent separation method — after the execution of  $o_{41}$ .**

An operation history is an uninterrupted operation set. According to Theorem 1, the causal relationship between two arbitrary operations in the operation history can be determined exactly by tracing its direct causal relationships. However, it has low efficiency. In worst case, it should traverse all the history operations in operation history.

Our solution is based on the following observations.

Every time before executing a remote operation  $o_r$ , all the earlier executed operations that are concurrent with it are separated out in  $set_{o_r}^c$  in advance (refer to chapter 4.2). If this result is cached, things will become much easier: given two operations  $o_a$  and  $o_b$  in the operation history, suppose  $o_a$  was executed earlier than  $o_b$ , to determine their causal relationship, we just need to check whether  $o_a$  belongs to  $set_{o_b}^c$ .  $o_a$  is concurrent with  $o_b$  if it belongs to  $set_{o_b}^c$ . Otherwise, it must be causally preceding  $o_b$ .

The number of operations that an operation is concurrent with could be very large. Accordingly, the size of  $set^c$  could also be very large. Fortunately, we do not need to keep in cache for each history operation a full version of  $set^c$ . Given an arbitrary history operation  $o$ , suppose  $o_a$  and  $o_b$  are two operations in  $set_o^c$ , and suppose  $o_a$  and  $o_b$  are generated at the same site,  $o_a$  is generated earlier than  $o_b$ . Consider what happens if we remove  $o_b$  from  $set^c$  before putting it in cache. When determining the causal relationship between  $o_b$  and  $o$ , we can first confirm that either  $o_b \parallel o$  or  $o_b \rightarrow o$ , for  $o_b$  is synchronized earlier than  $o$ ; noticing that  $o_a$  is contained in cached  $set_o^c$ , we can further conclude  $o_a \parallel o$ ; now consider  $o_a$  and  $o_b$  are coming from the same cooperating site,  $o_a \rightarrow o_b$ , it should not be difficult to find that  $o_b \parallel o$ . To sum up, the causal relationship between  $o_b$  and  $o$  can still be efficiently and exactly determined. In general, if there are several operations in  $set_o^c$  that come from the same cooperating site, we can safely remove all but the earliest generated one of them before caching it.

Following is a formal description of our solution. Here,  $SITE(o)$  represents the identifier of the site at which operation  $o$  was generated.  $GEN\_SEQ(o)$  represents the generation sequence number of  $o$  at  $SITE(o)$ .  $E\_SEQ(o)$  represents the execution sequence number of  $o$  at current cooperating site.

---

**Algorithm 2** Build\_Cache( $set_{o_r}^c$ ):  $cache_{o_r}^c$

---

```

1:  $cache_{o_r}^c \leftarrow \phi$ 
2: for all  $o_h$  in  $set_{o_r}^c$  do
3:   if  $cache_{o_r}^c[SITE(o_h)]$  is undefined then
4:      $cache_{o_r}^c[SITE(o_h)] \leftarrow GEN\_SEQ(o_h)$ 
5:   end if
6:   if  $cache_{o_r}^c[SITE(o_h)] > GEN\_SEQ(o_h)$  then
7:      $cache_{o_r}^c[SITE(o_h)] \leftarrow GEN\_SEQ(o_h)$ 
8:   end if
9: end for
10: return  $cache_{o_r}^c$ 

```

---

After procedure *Concurrent\_Separation* has been successfully executed, its result  $set^c$  will be passed along to procedure *Build\_Cache*. The procedure *Build\_Cache* will remove redundancy from  $set^c$  and return a condensed version of it  $cache^c$  for cache use.  $cache^c$  is an associative vector indexed by cooperating site identifier. *Build\_Cache* will create an item in  $cache^c$  for a cooperating site  $s$  (an item with site  $s$ 's identifier as the key) when it find in  $set^c$  an operation that was generated at  $s$  (line 3-5). Given a cooperating site  $s$ ,  $cache^c[s]$  records the generation sequence number of the earliest generated one of the operations in  $set^c$  that are generated at  $s$  (line 6-8).

---

**Algorithm 3** Detect\_Causality( $o_x, o_y$ )

---

```

1: if  $E\_SEQ(o_x) < E\_SEQ(o_y)$  then
2:   if  $o_y$  is local operation then
3:     return  $o_x \rightarrow o_y$ 
4:   else
5:     if  $cache_{o_x}^c[SITE(o_x)]$  is undefined or
6:        $cache_{o_x}^c[SITE(o_x)] > GEN\_SEQ(o_x)$  then
7:       return  $o_x \rightarrow o_y$ 
8:     else
9:       return  $o_x \parallel o_y$ 
10:    end if
11:  else
12:    if  $o_x$  is local operation then
13:      return  $o_y \rightarrow o_x$ 
14:    else
15:      if  $cache_{o_y}^c[SITE(o_y)]$  is undefined or
16:         $cache_{o_y}^c[SITE(o_y)] > GEN\_SEQ(o_y)$  then
17:        return  $o_y \rightarrow o_x$ 
18:      else
19:        return  $o_y \parallel o_x$ 
20:      end if
21:    end if

```

---

To determine the causal relationship between two history operations  $o_x$  and  $o_y$ , in procedure *Detect\_Causality*, we first determine the synchronization order of these two operations (line 1). If  $o_x$  is synchronized before  $o_y$ , as an example, there must be  $o_x \parallel o_y$  or  $o_x \rightarrow o_y$ . If  $o_y$  is a local operation, it is out of question that  $o_x$  must be causally preceding  $o_y$  (line 2-3). Otherwise, we check  $o_y$ 's related causality cache data  $cache_{o_y}^c$  to see whether there is some operation that is generated at the same site of  $o_x$  earlier than  $o_x$  and is concurrent with  $o_y$  (line 5). If there is, it must hold that  $o_x \parallel o_y$  (line 8), otherwise,  $o_x \rightarrow o_y$  (line 6).

As an illustration, still consider the scenario in Figure 1. Continue with previous analysis: suppose a new remote operation  $o_{41}$ , whose direct causal vector is  $[(1, 1), (3, 2)]$ , is received at site 3 shortly after the execution of  $o_{22}$  on that site.

Before  $o_{41}$ 's execution, procedure *ConcurrentSeparation* is first invoked to select all the history operations that are concurrent with  $o_{41}$  from operation history. In the previous subsection, we have analyzed the execution process of *ConcurrentSeparation*( $o_{41}$ ) and concluded that three operations  $o_{22}$ ,  $o_{12}$  and  $o_{21}$  will be returned as the result. Later, the result operation set is passed to procedure *BuildCache*. *BuildCache* scans the three operations  $o_{22}$ ,  $o_{12}$  and  $o_{21}$  in order. The variable  $cache_{o_{41}}^c$  is first initialized with empty set. After  $o_{22}$  has been scanned, according to line 4 in the procedure, this variable  $cache_{o_{41}}^c$  is updated to  $\{cache_{o_{41}}^c[s_2] = 2\}$ . Then  $o_{12}$  is scanned and for the same reason,  $cache_{o_{41}}^c$  is updated to  $\{cache_{o_{41}}^c[s_2] = 2; cache_{o_{41}}^c[s_1] = 1\}$ . And finally  $o_{21}$  is scanned. Because the current value of  $cache_{o_{41}}^c[s_2]$  is 2, that is,  $cache_{o_{41}}^c[s_2] > GEN\_SEQ(o_{21})$ , according to line 7 in the procedure, the value of  $cache_{o_{41}}^c[s_2]$  should be updated to 1. The final status of  $cache_{o_{41}}^c$  is  $\{cache_{o_{41}}^c[s_2] = 1; cache_{o_{41}}^c[s_1] = 1\}$ . It is returned as a condensed version of  $o_{41}$ 's concurrent preceding operation list for cache use.

$cache_{o_{41}}^c$  is put into the causality cache after it is computed. Later if it is needed at site 3 to compute the causal relationship between  $o_{41}$  and one of the operations  $o_{11}$ ,  $o_{12}$ ,  $o_{21}$ ,  $o_{22}$ ,  $o_{31}$  and  $o_{32}$ , which are all operations executed prior to  $o_{41}$  at site 3, the cached value  $cache_{o_{41}}^c$  can be used. Consider the causality detection between  $o_{41}$  and  $o_{31}$ . Because  $cache_{o_{41}}^c[s_3] = 0$ , according to the previous analysis, we can ascertain that  $o_{31} \rightarrow o_{41}$ . Again consider the causality detection between  $o_{41}$  and  $o_{22}$ . Because the value of  $cache_{o_{41}}^c[s_2]$  is 1, which is smaller than the generation sequence number of  $o_{22}$  at site 2, we can safely deduce that  $o_{22} \parallel o_{41}$ . It is easy to see that, with causality cache, the causal relationship between two arbitrary history operations in operation history can always be determined within a constant time.

## 5. DISCUSSION

In this section, we will analyze some important features of our approach. We will prove that our approach can be well adapted to wiki like large-scale dynamic collaboration environments.

We first consider the timestamp size. Direct causal vector timestamp is a kind of dynamically sized timestamp. It is not pre-allocated for each potential participants an item in the direct causal vector timestamp, so collaboration participants will not contribute to the size of direct causal vector timestamps until she/he begins editing the shared object. Thus, in our approach, it should work well in large-scale open collaboration environments.

A most critical feature of our approach is that, after a participant stops editing the shared object, she/he will also automatically and gradually stop contributing to the size of direct causal vector timestamps. Refer to Figure 4 as an illustration. There are totally three cooperating site 1, 2 and 3. After site 1 begins to edit the shared object, it begins to affect the cost of direct causal vector timestamp in the real-time group editor system.  $o_{22}$  is an operation that is generated after site 1 has begun editing the shared object. Its direct causal vector timestamp is  $[(1, 1), (2, 1)]$ . There is a timestamp item  $(1, 1)$  corresponding to site 1. After site

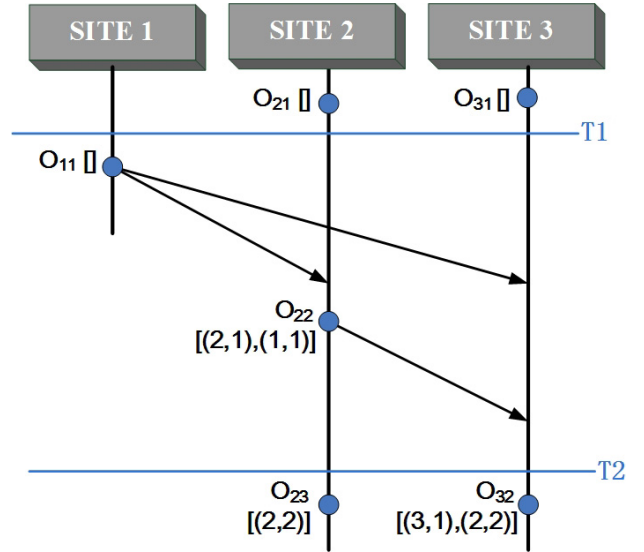


Figure 4: Direct causal vector approach and dynamic collaboration environment.

1 has stopped editing for an enough long time, at  $T2$ , when the operation  $o_{22}$ , which is causally dependent on  $o_{11}$ , has been propagated to and synchronized on all the cooperating sites, it can be proved that none of the operations generated afterwards have a timestamp item corresponding to site 1 in their direct causal vector timestamp. This will hold until site 1 is again active editing the shared object. It could be seen in Figure 4 that, in the direct causal vector timestamps of  $o_{23}$  and  $o_{32}$ , which are operations generated at site 2 and site 3 respectively after time  $T2$ , there is no timestamp item related with site 1.

It is not difficult to conclude that the size of the direct causal vector timestamp of an operation  $o$  approximates the number of participants active at editing the shared object recently before the generation of operation  $o$ .

Now we turn to the complexity of our algorithms. Assume that the number of participants concurrently editing the shared object never exceed  $L$ .

Our approach requires that four special data structures be kept at each cooperating site: DCH, DCG, LF mapping and causality cache. The size of DCH is never larger than  $L$ ; the storage complex of DCG is  $O(nL)$ ; the storage complexity of LF mapping is  $O(n)$ ; the storage complexity of causality cache is  $O(nL)$ , where  $n$  represents the size of the operation history. To sum up, the average storage cost of our approach (total storage cost/size of HB) is  $O(h)$ .

Our algorithms can detect for a remote operation whether it has become causally ready in  $O(L)$  time. The time complexity of algorithm *ConcurrentSeparation* is  $O(h + L^2)$ , where  $h$  is the number of history operations concurrent with the input remote operation. With the help of causality cache, algorithm *DetectCausality* can detect the causal relationship between two arbitrary history operation in  $O(1)$  time.

It is obvious that the complexity of our approach is not correlative with the total number of collaboration participants. Our approach can work well in collaboration environments of any scale, as long as the concurrency degree is not too high, which means that it never happens that a huge

number of people edit the shared object concurrently at a same time. As to our experience, in existing wikis, it rarely happens that more than ten people concurrently edit the same wiki page. Take Wikipedia as an example. According to the statistics from wikipedia web site [27], it seldom happens that a normal wikipedia page is edited more than ten times within less than an hour. Thus, our approach should be able to work efficiently in wiki collaboration environments.

We have notice that by carefully controlling the synchronize order of remote operations at each collaborating sites, the size of direct causal vector timestamp can be kept small even in a highly concurrent environment. This indicates the possibility of adapting our current direct causal vector approach to collaboration environments with much higher concurrency. Due to space limitation, we will not discuss it further. It should be part of our future work.

Notice that, unlike other solutions, as are discussed in the next section, our approach does not place any constraint on the network reliability, the network performance, the network topology or the mode in which people collaborating with each other.

## 6. RELATED WORKS

Various methods have been proposed to compress the size of vector clocks. On one extreme, methods have been proposed in [14, 20] to reduce the timestamp data to a single integer, but this comes at the cost of an increased computational overhead for the calculation of the vector clocks assigned to events, which is so large that it will slow down the distributed computation in an unacceptable way. Therefore, these methods are mainly applicable for a trace-based off-line analysis of the causality relation.

Other methods have been proposed to dynamically compress the size of vector clock[11, 16]. These method are all based on the following observation: even though the number of processes is large, only a few of them will interact with each other frequently by passing messages. In Singhal-Kshemkalyani's technique[16], vector clock compression is achieved by carrying in each message only those entries of the vector clock that have been updated after the previous communication between any pair of process. Ratner et al.[11] employs another method: for each entry of the vector clock, it tries to achieve consensus on a value to be subtracted and an entry will be removed once it is subtracted to 0. The main problem with these methods is that the size of the message timestamps is still linear in  $N$  (the number of communicating processes) in the worst case, as long as the collaboration is active enough. Furthermore, the Singhal-Kshemkalyani technique requires that communication channels be FIFO, while Ratner et al.'s method is only adapted to a well connected network.

The method proposed in [21] requires that the communication links between processes be static and known ahead of time. It is impossible in a large-scale collaborative environment.

Sun and Cai proposed an OT based method[6]. It is enforced that every operation generated at a certain cooperating site should be sent to the central site, transformed there against all the previously arrived concurrent operations, even if there exist some conflicts, and propagated to other cooperating sites in its transformed form, rather than original form. Thus, the method that can be applied to han-

dle conflict operations is constrained. The MVSD method proposed in [12] no longer applies. This method also suffers from single-point failure. Cooperating sites can not cooperate with each other if the central site is not accessible even if they themselves are well connected. Moreover, with this method, it is difficult to provide good performance for all the cooperating sites.

There also exist some OT algorithms that do not use the vector clock[10, 13, 19]. In NICE[13], a central site is employed to ensure that every operation propagated to a cooperating site  $i$  has been transformed against all the concurrent operations  $i$  received before. It suffers from similar deficiencies as the method proposed by Sun and Cai[6].

Both SOCK4[19] and TIBOT[10] use a scalar clock. In SOCK4, every operation is assigned a continuous serial number that is obtained from a global sequencer. It is required that a local operation should not be broadcast until all the operations with lower serial number have been transformed against it. In TIBOT, every site maintains a linear logical clock. All clocks are initialized to a common value and take the same sequence of values. The period between two consecutive clock ticks is defined as a time interval. It is required that a local operation should not be broadcast until all the operations generated in earlier time intervals have been transformed against it. Due to these constrains, in both SOCK4 and TIBOT, collaboration will be suspended even if only one of the clients is disconnected. Thus, neither of them can be used in large-scale collaborative environments based on Internet like unreliable network like wiki. The collaboration mode is also constrained. The collaboration provided by SOCK4 and TIBOT cannot be partial, that is either all sites collaborate or each one works separately.

## 7. CONCLUSION

In this paper we propose a new approach for capturing causality in real-time group editors. Our approach is based on a new kind of logical clock called DCV. Compared with other approach, it is easier to adapt to wiki like large-scale, highly dynamic, and unreliable collaboration environments.

Our approach enables the support of real-time group editing in wiki like applications, which should be beneficial for such category of applications to better support users' collaboration activities. Supporting large-scale collaborations over a wide-area network can help to gain more usage and visibility for Computer-supported Cooperative Work (CSCW).

Our approach currently do not support collaboration environments with high concurrency. We have seen the possibility of adapting it to support such collaboration environments. It should be one of our future work to delve into it in more detail. There exist a lot of real-time group editor related technologies that could help people better collaborate with each other, such as telepointer, radar view and etc. How to apply these technologies to wiki like applications? What are the effects? These should also be part of our future work.

## 8. REFERENCES

- [1] A. Kittur, B. Suh, B. A. Pendleton, and E. H. Chi. He says, she says: conflict and coordination in wikipedia. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 453–462, New York, NY, USA, 2007. ACM Press.

- [2] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 399–407, May 1989.
- [3] C. Gutwin and S. Greenberg. The importance of awareness for team cognition in distributed collaboration. *Team Cognition: Understanding the Factors that Drive Process and Performance*, APA Press, pages 177–201, 2004.
- [4] C. M. Hymes and G. M. Olson. Unblocking brainstorming through the use of a simple group editor. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 99–106, November 1992.
- [5] C. Sun and C. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 59–68, December 1998.
- [6] C. Sun and W. Cai. Capturing causality by compressed vector clock in real-time group editors. *International Parallel and Distributed Processing Symposium - Symposium Volume*, pages 59–67, 2002.
- [7] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-human Interaction*, 5(1):63–108, March 1998.
- [8] D. Li and R. Li. Preserving operation effects relation in group editors. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 457–466, November 2004.
- [9] D. Li and R. Li. A landmarkbased transformation approach to concurrency control in group editors. *ACM GROUP'05*, November 2005.
- [10] D. Li, R. Li, and C. Sun. A time interval based consistency control algorithm for interactive groupware applications. *Proceedings of the Parallel and Distributed Systems*, pages 429–440, 2004.
- [11] D. Ratner, P. Reiher, and G. Popek. Dynamic version vector maintenance. *Technical Report CSD-970022*, 1997.
- [12] D. Sun, S. Xia, C. Sun, and D. Chen. Operational transformation for collaborative word processing. *Proceedings of the ACM Conference on Computer Supported Cooperative Work CSCW'04*, pages 437–446, 2004.
- [13] H. Shen and C. Sun. Flexible notification for collaborative systems. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 77–86, December 2002.
- [14] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. *Proc. of 10th Int. Conference on Distributed Computing Systems*, pages 134–141, May 1990.
- [15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, pages 558–565, 1978.
- [16] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing letters*, pages 47–52, August 1992.
- [17] M. Suleiman, M. Cart, and J. Ferrie. Serialization of concurrent operations in distributed collaborative environment. *ACM GROUP'97*, pages 435–445, November 1997.
- [18] N. Gu, J. Yang, and Q. Zhang. Consistency maintenance based on the mark & retrace technique in groupware systems. *ACM GROUP'05*, pages 264–273, November 2005.
- [19] N. Vidot, M. Cart, J. Ferrie, and M. Suleiman. Copies convergence in a distributed realtime collaborative environment. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 171–180, December 2000.
- [20] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributing Computing*, pages 149–174, 1994.
- [21] S. Meldal, S. Sankar, and J. Vera. Exploring locality in maintaining potential causality. *Proc. of 10th ACM Symposium on Principles of Distributed Computing*, pages 231–239, 1991.
- [22] T. Prante and C. Magerkurth. Developing csw tools for idea finding — empirical results and implications for design. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 106–115, 2002.
- [23] StrategyWiki. [http://strategywiki.org/wiki/Main\\_Page](http://strategywiki.org/wiki/Main_Page).
- [24] WikiNews. <http://en.wikinews.org/>.
- [25] Wikipedia. What Is Wiki? <http://www.wiki.org/wiki.cgi?WhatIsWiki>.
- [26] Wikipedia. <http://en.wikipedia.org/>.
- [27] wikistat. <http://www.wikipedia.org/wiki/Wikipedia:Statistics>.
- [28] WikiTravel. [http://wikitravel.org/en/Main\\_Page](http://wikitravel.org/en/Main_Page).