A Novel Strategy for Pipelining Restricted Vector Clocks in Distributed Systems

Hwa-Chuan Lin and SingLing Lee

Department of Computer Science and Information Engineering

National Chung Cheng University, Chiayi 62107

Taiwan, Republic of China

Abstract

The vector clocks (VC) mechanism has been employed to solve a wide variety of problems in many applications, provided that the bit size of each component in the VC is always sufficient. However, this may lead to an extremely large overhead in communication bandwidth and storage as the number of processes gets to be very large. Otherwise, the VC resetting procedures must be deployed and invoked carefully, which results in a restricted vector clock (RVC) property. The PVC employs two different representations to denote the meanings of VC in different cycles, so that the monotonic characteristic of VC is maintained for the execution of any applications.

Keywords: Clock overflow, distributed systems, timestamp, vector clocks.

1 Introduction

Due to lack of a standardized global time for distributed systems, using logical time is a powerful concept to maintain causal relations among events [11, 15]. Traditional vector clocks (VC), introduced independently by both Fidge [10] and Mattern [12], is a useful mechanism for understanding the causal relation between events in distributed systems. The concept of VC has been extensively used in many applications, such as distributed debugging [14], detecting global predicates [6, 12], recovery [2, 14], snapshot [8], causal multicast (or broadcast) [1, 4, 5, 7].

1.1 Motivation

Most protocols using VC take for granted that the bit size of each component in the VC is always sufficient for running an application. In other words, they neglect the clock overflow problem. Because VC can grow without bounds, applications that use VC may require unbounded space. Thus, in realistic implementation, applications using VC may have a serious problem with overhead in message-passing and overhead in storage as the number of processes gets to be very large. This is exacerbated by the fact that the more bits used in each element of the vector clock, the more overhead will be needed.

Invoking the VC resetting protocol before any clock overflow happening is another way to avoid the VC overflow problem [17], although this restricts the VC property in the course of executing an application. In addition, the VC may be reset to be zero more frequently when running the protocol, which may violate the correctness if the VC resetting is not carefully handled.

Another method is to find a bounded number of clock values for VC, such as resettable vector clocks [3] and bounded timestamps [16]. The reusability of timestamps is one of their characteristics, however, it is time-consuming for finding and maintaining the

bounded timestamps.

In the proposed pipelining vector clocks (PVC), we dynamically employ the following two different representations for VC: *magnitude representation* and *signed 2's complement representation* [13]. Our PVC has the same power as the traditional VC if the bit size for each element in vector clocks is sufficient, but it avoids pitfalls above-mentioned in traditional VC.

1.2 Basic Idea

Suppose that there are three bits b_2, b_1, b_0 used for the clock. We list two different representations for bits b_2, b_1 and b_0, as shown in Figure 1. One is *magnitude representation*, and the other is *signed 2's complement representation*. In magnitude representation, these binary bits (from $000, 001, 010, 011, 100, 101, 110$ to 111 in succession) can denote values from 0 to 7, respectively. However, these bits can also denote values from $0, 1, 2, 3, -4, -3, -2,$ to -1, respectively, in signed 2's complement representation.

Note that b_2 is a phase bit (i.e., sign bit) in signed 2's complement representation, but it is a most significant bit in magnitude representation. As an example, 101_2 denotes the values 5_{10} in magnitude representation and -3_{10} in signed 2's complement representation, respectively.

The example of Figure 1 gives a good representation of the basic idea of our proposed PVC. As Figure 1 indicates, there are several monotonic increasing-value cycles, as denoted by arrows; the arrows on top (such as the arrow indexed by y'_1, y'_2) indicate the monotonic duration of a clock in signed 2's complement representation. On the other hand, arrows at the bottom (such as the arrow indexed by y_1, y_2) indicate the monotonic duration of clock in magnitude representation. To synchronize the process phases, in Section 5.2 we introduce the concept of *virtual synchronization*, which simultaneously

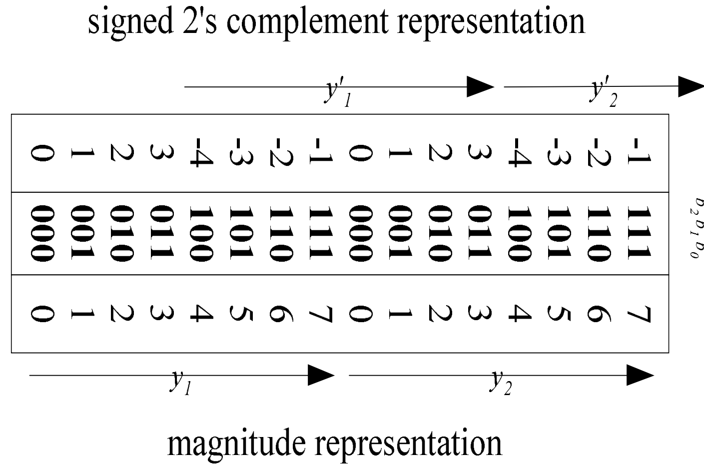

Figure 1: Example of two different meanings in unsigned-magnitude representation and signed 2's complement representation.

synchronizes all process phases into the same phase.

1.3 Contribution

This work presents the first algorithm that implements VC by using two different representations. To the best of our knowledge, only one representation (i.e., the unsigned-magnitude representation) is used to denote the meanings of traditional VC.

As Yen [17] pointed out, prior knowledge about process behavior is necessary for finding an adaptive bit size for each element of VC. As the bit size for each element of VC is determined, the proposed method has the following salient features:

1. The proposed PVC can make the protocols using VC highly scalable. In our proposed PVC, we use two representations (i.e., *magnitude representation* and *signed 2's complement representation*) to denote the meanings of VC in different cycles (i.e., *zero-one cycle* and *one-zero cycle*) such that the VC property is valid during the execution of any short-term or long-term applications using PVC.

2. Our scheme does not have the drawback of clock overflow; it can avoid the trouble of determining a triggering condition for initiating the VC resetting protocol.

2 Related Work

Yen et al. proposed a clock resetting protocol that is invoked carefully by satisfying the predefined threshold defined in [17]. In that method, each process must carefully count the number of events and invoke the reset protocol immediately when the counting value reaches a predefined threshold. After executing the clock resetting protocol, all processes reset their VCs, so the condition of clock overflow can be prevented in any process. Clearly, with Yen's protocol, there are many timestamping phases that occur in an application, implying that the monotonic characteristic of the VC is maintained only in each timestamping phase. In other words, the monotonic characteristic of VC is invalid even for two adjacent timestamping phases. Hence, the monotonic property of the VC in Yen's protocol is limited to a timestamping phase, and this is why it is called restricted vector clock (RVC) property. Furthermore, application messages are not allowed to be transmitted during the execution of Yen's resetting protocol.

Singh in [16] proposed a technique of bounded timestamps, supposing that values x_1,..., x_n are used in round-robin fashion. Process P_i can reuse a timestamp value x_k when all old messages with value x_k in the ith element of VC have been delivered in all the processes of the system. As mentioned above, it is impossible to find an adaptive bit size for each element of VC without any prior knowledge about process behavior. Clearly, it is time-consuming to find and monitor the bounded timestamps.

Based on the observation that applications can be structured in phases and track causality only within a bounded number of adjacent phases, Arora et al. [3] proposed a bounded-space resettable vector clocks with a bounded number of adjacent phases.

Resettable vector clocks, which can be viewed as a variation of [17], allow nonblocking reset to be performed by a process when it reaches the bound of its local clock. However, not all applications using VC can be directly substituted by bounded-space resettable vector clocks because the total number of events may be unbounded and unpredictable [3].

3 System Model

Letting V_i be the VC of process P_i, and supposing that there are n processes running an *application* in the system, Charron-Bost [9] has proven that the size of V_i maintained by each process P_i must be at least n (i.e., $V_i[1..n]$) in order to capture the causal relation between events. In addition, let $V_i[j]$ denote the jth element of V_i. Initially, the n-element scalar vector clocks V_i of every process is set to zero (i.e., $\forall j : V_i[j] \leftarrow 0$), which is denoted as $V_i \leftarrow 0$ for the sake of simplicity.

There are three kinds of events in a distributed system: the event of sending a message, the event of delivering the message, and the internal event. Internal events represent the local computation events of processes. Generally, $V_i[i]$ is incremented by a value of one each time process P_i sends a message, process P_i receives a message, or an internal event occurs in P_i. Note that each sending message m by a process P_i is timestamped with the VC of the process P_i at the time the event occurs, which is called m's timestamp.

Assuming that an application message m arrives at the protocol at process P_i, we say that process P_i receives and delivers m until the protocol forwards m up to the application. Like [1, 5, 17], we suppose that the communication between any two adjacent processes is reliable and FIFO-ordered. Notice that many network transport protocols support the FIFO order property. We also assume that message transmission delays are arbitrary but

finite. The system model assumed here is quite common in the context of distributed computing systems.

Lamport defines the "happen-before" relation (denoted by "$\rightarrow$") to solve the difficulty of determining the causal ordering of events occurring in systems [11]. Letting $send_i(m)$ and $deliv_j(m)$ represent the events of the sending m from P_i and delivery of m by P_j, respectively, the "happen-before" relation is as follows:

1. For any two events e and e', if event e occurs at process P_i before e', then $e \rightarrow e'$.
2. For any two events e and e', if event e is the event $send_i(m)$ and event e' is the event $deliv_j(m)$, then $e \rightarrow e'$.
3. If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.

4 Pipelining Vector Clock

4.1 Pipelining Vector Clock Property

A close look at Figure 1 will reveal that the monotonic characteristic of a clock can be maintained if we can properly denote the meanings of a clock, such as cycles y_1, y_1', y_2, y_2' in the figure.

Generally, let the leftmost significant bit of a clock be a special bit, called the *phase-bit*, which records the process's phase when timestamping. Taking Figure 1 for example, let b_2 denote the phase bit, and the other bits (i.e., b_1, b_0) are used for the scalar clock. Clearly, the phase bit is altered when scalar clock overflows.

Note that P_i is in phase 0 (phase 1) when the phase bit of P_i is 0 (1). In order to synchronize all the process phases into the same phase, *virtual synchronization* (introduced in Section 5.2) will be invoked if necessary. For the sake of simplicity, we assume that

there are two phases in the system: phase 0 and phase 1.

Definition 1 *Let a zero-one cycle, y, be the duration composed of two adjacent phases such that phase 0 is before phase 1. In contrast, a one-zero cycle, y', has phase 1 before phase 0.*

For example, cycles y_1 and y_2 shown in Figure 1 are *zero-one cycles*. Note that each pair of adjacent cycles (y_i, y'_i) has a common duty phase 1, and each pair of adjacent cycles (y'_i, y_{i+1}) has a common duty phase 0.

Property 1 *In PVC, each pair of adjacent cycles (y_i, y'_i) has a common duty phase 1, and each pair of adjacent cycles (y'_i, y_{i+1}) has a common duty phase 0.*

We use "$\|$" (i.e., concatenation) to define a composite timestamp $\top_a$ of event a, which is composed of a phase bit $\wp_a$ and a scalar timestamp T_a (i.e., $\top_a = \wp_a \| T_a$). Suppose that $<_u$ is a *less-than* relation in magnitude representation. Let us assume that $<_s$ is a *less-than* relation in signed 2's complement representation. Then, we obtain the following properties of PVC.

Property 2 *Let $\mathcal{E}^y$ be the set of all the events occurring in a zero-one cycle y and let $\mathcal{E}^{y'}$ be the set of all the events occurring in one-zero cycle y'.*
(a) From the viewpoint of the zero-one cycle, y, $\forall a, b \in \mathcal{E}^y : a \rightarrow b \Leftrightarrow \top_a <_u \top_b$.
(b) From the viewpoint of the one-zero cycle, y', $\forall a, b \in \mathcal{E}^{y'} : a \rightarrow b \Leftrightarrow \top_a <_s \top_b$.

4.2 Requirements of PVC

The goal of PVC's requirements is that the sending and corresponding reception of each message must be kept in the same cycle. Let us assume that $send_i(m)$ and $deliv_j(m)$ represent the events of sending m from P_i and delivery of m by P_j, respectively. Let $y(t)$

denote the phase t of *zero-one cycle* y, where $t \in \{0, 1\}$. Similarly, the same observation applies to the *one-zero cycle* y'. The requirements for PVC are as follows.

R1. All messages sent in the phase 0 of *zero-one cycle* y from process P_i to process P_j can be received by P_j in the duration of *zero-one cycle* y. In other words, if $send_i(m) \in \mathcal{E}^{y(0)}$, then $deliv_j(m) \in \mathcal{E}^{y}$.

R2. All messages sent in the phase 1 of *one-zero cycle* y' from process P_i to process P_j can be received by P_j in the duration of *one-zero cycle* y'. In other words, if $send_i(m) \in \mathcal{E}^{y'(1)}$, then $deliv_j(m) \in \mathcal{E}^{y'}$.

5 The Implementation

Granted that the system model described in Section 3 is used in the implementation. Let $\mathcal{N}(P_i)$ be the set of processes with a FIFO-ordered communication channel connected to P_i; and let Ψ_i denote P_i's phase bit and V_i denote P_i's scalar VC. Initially, Ψ_i and V_i are initialized to be zero. As noted before, each process P_i can operate in phase 1 or phase 0. Suppose that the system begins with phase 0. Our procedure consists of three major parts: the sending message, receiving message, and virtual synchronization parts.

5.1 The Modules of Sending and Receiving Application Messages

First, we introduce actions at process P_i when an application message m is to be sent. When sending a message m, process P_i attaches scalar VC V_i and P_i's phase Ψ_i to m, as denoted by $m.T$ and $m.\Psi$, respectively.

Second, upon first receiving message m, process P_i checks the delivery condition to decide what action should be taken next. If R1 listed in Section 4.2 is satisfied, P_i performs

the actions: for all $k \neq i$: $\Psi_i \| V_i[k] \leftarrow max_u(\Psi_i \| V_i[k], m.\Psi \| m.T[k])$. Otherwise, if it satisfies R2, P_i performs the actions: for all $k \neq i$: $\Psi_i \| V_i[k] \leftarrow max_s(\Psi_i \| V_i[k], m.\Psi \| m.T[k])$. Afterwards, advance $V_i[i]$. The actions of sending the application message and receiving the application message are as follows.

1. When an internal or sending event e occurs at process P_i do:

   (a) $V_i[i] \leftarrow V_i[i] + 1$;

   (b) If $V_i[i] = 0$ and $S_i = normal$ then

   $S_i \leftarrow trans$;

   $\Psi_i \leftarrow \neg \Psi_i$;

   $V_i \leftarrow 0$;

   send *NewPhase_req* to all $P_k \in \mathcal{N}(P_i)$;

   (c) If sending an application message m, process P_i attaches V_i and Ψ_i to the outgoing message m, as are denoted by $m.T$ and $m.\Psi$, respectively.

2. When an application message m sent by process P_j arrives at process P_i do:

   (a) If ($\Psi_i = 0$ and $m.\Psi = \Psi_i$) or ($\Psi_i = 1$ and $m.\Psi \neq \Psi_i$) then

   for all $k \neq i$: $V_i[k] \leftarrow max_u(V_i[k], m.T[k])$;

   elseif ($\Psi_i = 1$ and $m.\Psi = \Psi_i$) or ($\Psi_i = 0$ and $m.\Psi \neq \Psi_i$) then

   for all $k \neq i$: $\Psi_i \| V_i[k] \leftarrow max_s(\Psi_i \| V_i[k], m.\Psi \| m.T[k])$;

   (b) $V_i[i] \leftarrow V_i[i] + 1$;

   (c) If $V_i[i] = 0$ and $S_i = normal$ then

   $S_i \leftarrow trans$;

   $\Psi_i \leftarrow \neg \Psi_i$;

   $V_i \leftarrow 0$;

   send *NewPhase_req* to all $P_k \in \mathcal{N}(P_i)$;

Note that the *virtual synchronization* will be invoked when clock overflow occurs in any one of the processes due to advancing $V_i[i]$, such as Steps 1(b) and 2(c) in the above actions. In next subsection we present the details of *virtual synchronization*.

5.2 Virtual Synchronization Module

When clock overflow occurs in any one of the processes in the system, the "*virtual synchronization*" is initiated automatically, forcing all the processes to change their current phases and eventually enter *normal* status. Note that each process P_i can operate in either status: *normal* or *trans*. A process P_i is usually in *normal* status except during *virtual synchronization*. Taking Figure 2 for example, the thick line shown in the figure denotes the duration of *virtual synchronization*.

In order to synchronize the process phases, we need the following variables in each process P_i. Let S_i be a variable that records P_i's current status; and $N_{i,j}$ be a variable that records neighboring process P_j's status, where $P_j \in \mathcal{N}(P_i)$, currently known to P_i. Also, suppose that S_i and $N_{i,j}$ are initialized to *normal* for all i, j.

Any process P_i that first invokes the execution of the *virtual synchronization* is called an *initiator*, and it performs the following jobs (see Step (a) of Procedure *Receiving_NewPhase_req*): (1) set S_i to *trans* (i.e., $S_i \leftarrow trans$); (2) change phase (i.e., $\Psi_i \leftarrow \neg\Psi_i$); (3) reset scalar VC (i.e., $V_i \leftarrow 0$), and (4) send *NewPhase_req* to all $P_j \in \mathcal{N}(P_i)$.

On first receiving *NewPhase_req* from P_j, any process P_i operating in *normal* behaves like an initiator and sends out *NewPhase_req* to all its neighbors and then enters *trans*. In other words, the initiator plays a role in spreading the *NewPhase_req* over every communication channel, so that every process will receive *NewPhase_req* from its neighboring processes. Afterwards, P_i sets $N_{i,j}$ to *trans* (i.e., $N_{i,j} \leftarrow trans$; see Step (b) of Procedure

Receiving_NewPhase_req). After ensuring that $N_{i,k} = trans$ for all $P_k \in \mathcal{N}(P_i)$, P_i sets S_i to *normal* and $N_{i,k}$ to *normal* for all $P_k \in \mathcal{N}(P_i)$ (see Step (c) of Procedure *Receiving_NewPhase_req*). That is, P_i enters a *normal* stable status. When all processes are in *normal* status, that implies that the *virtual synchronization* is finally finished.

Receiving_NewPhase_req

When *NewPhase_req* sent from P_j arrives at P_i do:

(a) If $S_i = normal$ then

$S_i \leftarrow trans$;

$\Psi_i \leftarrow \neg\Psi_i$;

$V_i \leftarrow 0$;

send *NewPhase_req* to all $P_k \in \mathcal{N}(P_i)$;

(b) $N_{i,j} \leftarrow trans$;

(c) If $N_{i,k} = trans$ for all $P_k \in \mathcal{N}(P_i)$ then

$S_i \leftarrow normal$;

for all $P_k \in \mathcal{N}(P_i)$ do

$N_{i,k} \leftarrow normal$;

6 Example

Figure 3 shows an example diagram based on our method and the proposed model. Suppose that there are three processes (P_1, P_2, and P_3) in the system and each pair of processes is connected with each other by FIFO-ordered communication channels. Assume that the system is in phase 0 of *zero-one cycle y* (i.e., $\Psi_1 = \Psi_2 = \Psi_3 = 0$).

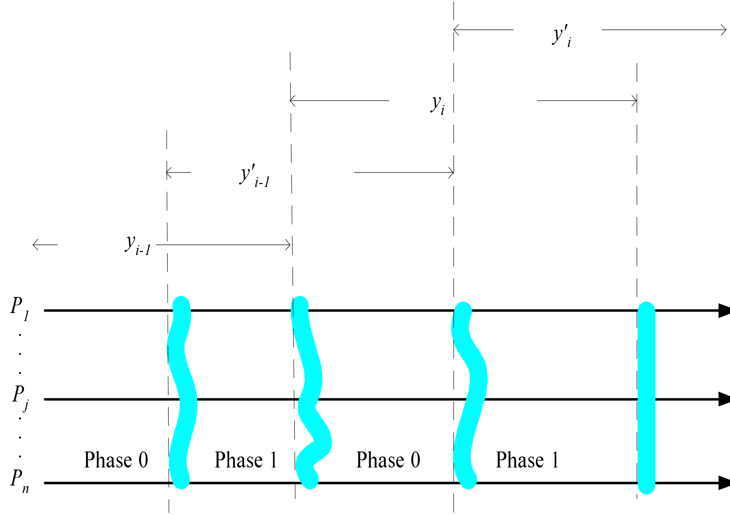

Figure 2: Example of phase diagram.

6.1 Virtual Synchronization

First, we introduce the effect of *virtual synchronization*. From the left part of Figure 3, we can see that P_1 invokes *virtual synchronization* at event $a1$. As noted in Section 5.2, we know that P_1 will change its phase, reset its scalar vector, and enters $trans$ status at that time. In other words, $\Psi_1 = 1$, $S_1 = trans$, and $V_1 = 0$. Upon receiving *NewPhase_req* sent by P_1, P_2 or P_3 will send out *NewPhase_req* to its neighboring processes. Afterwards, the system has $\Psi_1 = \Psi_2 = \Psi_3 = 1$, $S_1 = S_2 = S_3 = trans$, and $V_1 = V_2 = V_3 = 0$. In other words, the system is in the beginning of *virtual synchronization*.

From Section 5.2, we know that the beginning of *virtual synchronization* implies that the following actions have been performed in each process P_i: (1) set S_i to $trans$ (i.e., $S_i \leftarrow trans$); (2) change phase (i.e., $\Psi_i \leftarrow \neg\Psi_i$); (3) reset scalar VC (i.e., $V_i \leftarrow 0$). Clearly, events $a1$, $b1$ and $c1$ indicate the beginning of the *virtual synchronization*, as indicated by a dotted line, denoted $\overline{a1b1c1}$, which passes through P_1, P_2, and P_3, respectively.

As noted in Step (c) of Procedure *Receiving_NewPhase_req* (shown in Section 5.2), after ensuring all neighboring processes have reset their clocks, each process P_i must reset

S_i to *normal*. This implies that all processes are now back to *normal* status eventually after *virtual synchronization*. Figure 3 shows the dotted line passing through events a_3, b_2 and c_2, indicating the ending of *virtual synchronization*, denoted by $\overline{a3b2c2}$, which implies that all processes are back to *normal* status at that time.

Similarly, the other example of *virtual synchronization* shown in Figure 3 begins at events $a'1$, $b'1$ and $c'1$, represented by a dotted line passing those events (i.e., $\overline{a'1b'1c'1}$). Note that two *NewPhase_req* messages are sent from P_1 and P_3, indicating that P_1 and P_3 invoke the *virtual synchronization* at events $a'1$ and $c'1$, respectively. As mentioned above, we know that $V_1 = V_2 = V_3 = 0$, $S_1 = S_2 = S_3 = trans$, and $\Psi_1 = \Psi_2 = \Psi_3 = 0$ all occur at events $a'1$, $b'1$, and $c'1$. The other dotted line passing through events $a'3$, $b'2$ and $c'2$ indicates the ending of this *virtual synchronization*, denoted by $\overline{a'3b'2c'2}$.

Note that, due to the FIFO-ordered assumptions, no messages sent in phase 0 of the *zero-one cycle y* will cross the dotted line denoting the ending of corresponding *virtual synchronization* (i.e., $\overline{a3b2c2}$) shown in the figure. For example, messages m_0 and m_1 will not cross $\overline{a3b2c2}$. Similarly, all messages sent in phase 1 of *zero-one cycle y* will have been received by destination processes before the ending of the next *virtual synchronization* (i.e., $\overline{a'3b'2c'2}$). For example, messages m_2, m_3, m_4, m_5, m'_0, and m'_1 will not cross $\overline{a'3b'2c'2}$. Similarly, the same observation can be applied to the *one-zero cycle y'*.

6.2 Application Messages

Second, we would like to focus on the application messages shown in Figure 3. Suppose that an application message m sent by source process P_j arrives at destination process P_i. It is clear that we can classify the application messages into the following classes.

1. $\Psi_i = 0$ and $m.\Psi = \Psi_i$: See, e.g., messages m_0, m'_2, m'_3, m'_4, and m'_5. From Section 4.2, we know that messages sent to process P_i in phase 0 of the *zero-one cycle y*

can be received by P_i during the *zero-one cycle* y. In other words, those messages satisfy R1, as shown in Section 4.2.

2. $\Psi_i = 1$ and $m.\Psi \neq \Psi_i$: Such as m_1. Figure 3 shows that message m_1 is sent to process P_1 by P_3 in phase 0 of the *zero-one cycle* y. According to R1, we know that it can be received by P_1 during the *zero-one cycle* y. In other words, $send_3(m_1) \in \mathcal{E}^{y(0)}$ and $deliv_1(m_1) \in \mathcal{E}^y$ satisfy R1.

3. $\Psi_i = 1$ and $m.\Psi = \Psi_i$: Examples of this case are m'_0, m_2, m_3, m_4, and m_5. From R2 (shown in Section 4.2), we know that all messages sent in phase 1 of the *one-zero cycle* y' from a process P_j to a process P_i can be received by P_i during the *one-zero cycle* y'.

4. $\Psi_i = 0$ and $m.\Psi \neq \Psi_i$: Such as m'_1. From Figure 3, we know that $send_3(m'_1) \in \mathcal{E}^{y'(1)}$ and $deliv_1(m'_1) \in \mathcal{E}^{y'}$. Hence, m'_1 satisfies R2.

Note that no application messages are blocked during *virtual synchronization*. With Yen's method, messages like m_2, m_3, m'_2 and m'_3 are not allowed to be sent, and messages like m_4 and m'_4 must be buffered until after the resetting protocol.

7 Comparison

Assuming that the bit size of each component in the VC is always sufficient, VC has been used to solve a wide variety of problems in many applications; and we call this VC, *sufficient VC*. However, this may lead to a serious overhead in message-passing and storage.

With Yen's protocol [17], the VC resetting protocol must be invoked carefully, such as the dotted line shown in Figure 4(b), before any clock overflow happens in order to

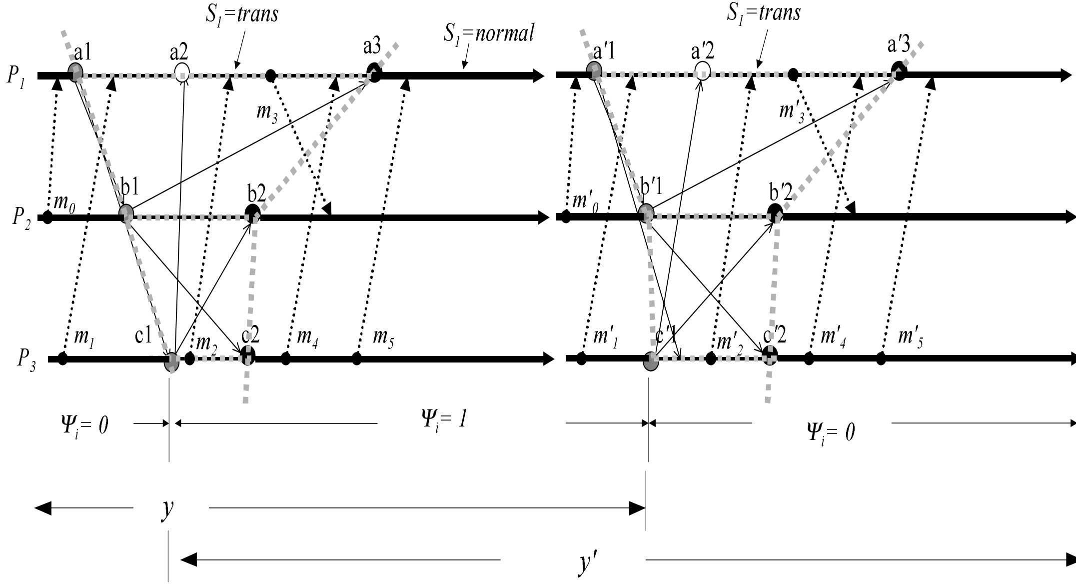

Figure 3: Example.

avoid the VC overflow problem. However, this can restrict the VC property (i.e., RVC) in the course of executing an application.

The dotted line shown in Figure 4(a) denotes the *virtual synchronization* by the PVC method. With *virtual synchronization* associated with the proposed two representations in PVC method, the VC property can be maintained and not be affected by clock overflow. Figure 4 indicates the major difference between PVC and RVC in time domain. Like Yen's method, there are many phases in our PVC method. However, no auxiliary rule is needed when we need to compare timestamps of events occurring in adjacent phases. Note that in Yen's method process P_i is not allowed to send any application messages when it is in *trans* status. Unlike Yen's method, the transmission of application messages are not inhibited during the execution of *virtual synchronization*.

Due to FIFO-order and reliable channel properties, which ensure the liveness property, $\forall m : send_i(m) \rightarrow deliv_j(m)$, implying that m sent by P_i will be delivered on P_j after a

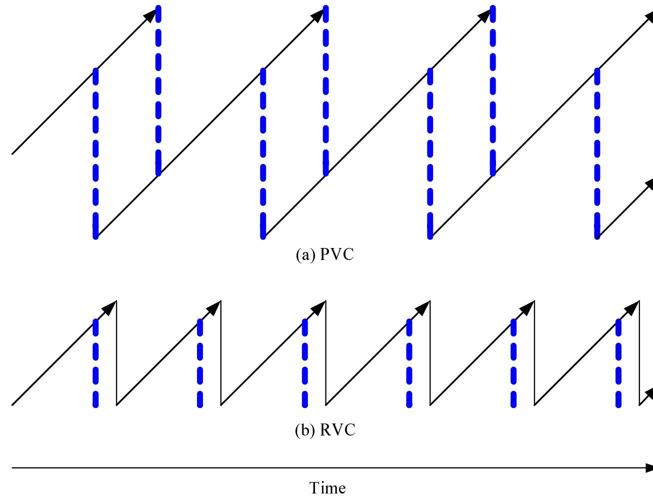

Figure 4: Time Diagram of PVC and RVC.

finite time. Furthermore, based on [17] and Property 2, we can determine that what is true for the RVC property in a timestamping phase of Yen's protocol is to a considerable extent also true for a *zero-one cycle* or *one-zero cycle* in our proposed PVC property. The strength of PVC is that it offers a mechanism to pipeline short-term VC segments into a long-term VC, so that clock overflow can be tolerated. In other words, we may also say that the PVC has the power to pipeline RVCs into *sufficient VC*.

8 Conclusion

Without assuming that the bit size of each component in the VC is always sufficient for running an application, we overcome the clock overflow problem, in which the VC's representation requires a tune-up when the clock overflows. In our proposed PVC, we use two representations (i.e., *magnitude representation* and *signed 2's complement representation*) to denote the meanings of VC in different cycles (i.e., *zero-one cycle* and *one-zero cycle*) such that the VC property is valid during the execution of any short-term or long-term

applications using PVC. The proposed PVC contributes to enlightening the power of traditional VC. Besides, unlike RVC, PVC relieves us from the difficult task of determining the triggering condition for the clock resetting protocol because the clock overflow can be tolerated.

References

[1] G. Anastasi, A. Bartoli and F. Spadoni, A reliable multicast protocol for distributed mobile systems: design and evaluation, IEEE Transactions on Parallel and Distributed Systems 12 (10) (2001) 1009-1022.

[2] S. Alagar and S. Venkatesan, An optimal algorithm for distributed snapshots with causal message ordering, Information Processing Letters 50 (1994) 311-316.

[3] A. Arora, S. Kulkarni, and M. Demirbas, Resettable vector clocks, Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing (2000) 269-278.

[4] R. Baldoni, A positive acknowledge protocol for causal broadcasting, IEEE Transactions on Computers 47 (12) (1998) 1341-1350.

[5] K. Birman and T. Joseph, Reliable communications in presence of failure, ACM Transactions on Computer Systems 5 (1) (1987) 47-76.

[6] Ö. Babaoğlu and K. Marzullo, Consistent global states of distributed systems: fundamental concepts and mechanisms, in S. Mullender (editor) Distributed Systems, Addison-Wesley, (1993) 55-96.

[7] K. Birman, A. Schiper, and P. Stephenson, Lightweight causal and atomic group multicast, ACM Transactions on Computer Systems 9 (3) (1991) 272-314.

[8] K. M. Chandy and L. Lamport, Distributed snapshots: determining global states of distributed systems, ACM Transactions on Computer Systems 3 (1) (1985) 63-75.

[9] B. Charron-Bost, Concerning the Size of logical clocks in distributed systems, Information Processing Letters 39 (1) (1991) 11-16.

[10] J. Fidge, Timestamps in message-passing systems that preserve the partial ordering, Proceedings of the 11th Australian Computer Science Conference 10 (1) (1988) 56-66.

[11] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM, 21 (7) (1978) 558-565.

[12] F. Mattern, Virtual time and global states of distributed system, in M. Cosnard et al. (editors) Parallel and Distributed Algorithms Elsevier, North-Holland, (1989) 215-226.

[13] M. Mano and C. Kime, Logical and computer design fundamentals, Second edition, Prentice Hall, (2000).

[14] M. Raynal, A. Schiper, and S. Toueg, The causal ordering abstraction and a simple way to implement it, Information Processing Letters 39 (1991) 343-350.

[15] M. Raynal and M. Singhal, Logical time: capturing causality in distributed systems, IEEE Computer 29 (2) (1996) 49-56.

[16] A. Singh, Bounded timestamps in process networks, Parallel Processing Letters 6 (2) (1996) 259-264.

[17] L.-H. Yen, T.-L. Huang, Resetting vector clocks in distributed systems, Journal of Parallel and Distributed Computing 43 (1997) 15-20.