Resetting Vector Clocks in Distributed Systems*

Li-Hsing Yen and Ting-Lu Huang

Department of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, Taiwan 300, R.O.C.

*This research was supported by the National Science Council, Taiwan, ROC, under grants NSC85-2213-E009-061.

Resetting Vector Clocks in Distributed Systems

Person to contact: Ting-Lu Huang

E-mail address: tlhuang@csie.nctu.edu.tw

Fax: (8863) 5724176, Tel: (8863) 5712121 Ext. 54725

Department of Computer Science and Information Engineering

National Chiao Tung University

Hsinchu, Taiwan 300, R.O.C.

Abstract

This paper establishes the necessary and sufficient condition for a correct clock resetting such that the functionality of vector clocks can be preserved. A clock reset protocol is presented with its applicability and limitation discussed. Our result indicates that for some applications, the potential of clock overflow can be completely prevented by carefully choosing the condition for initiating the clock reset protocol.

Key words: distributed systems, logical time, vector clocks, causal ordering, happened-before relation

a

b

m

n

P_i

$\mathcal{V}_i$ vee (calligraphic letter)

$\mathcal{T}$ T (calligraphic letter)

k

i

E^i

$sent(m)$

$recv(m)$

t_i

$\mathcal{E}$ E (calligraphic letter)

e_i

Φ phi (uppercase)

Σ sigma (uppercase)

ξ xi

ϕ phi (lowercase)

r

j

k

η eta

$L_{i,j}$

$\mathcal{L}_k^p$ L (calligraphic letter)

1 Introduction

No common system-wide clocks exist in distributed systems. Their absence makes it difficult to reason about temporal ordering of events occurring in these systems. Lamport [15] defined the *happened-before* relation which is a partial ordering on the set of system events. To realize this relation, Fidge [8] and Mattern [18] introduced the *vector clock* scheme, in which each event is timestamped with the value of the vector clock maintained by the process where the event occurs. By comparing the timestamps of events, we can determine the *causal relation* of events in the system.

It has been proven that, given n processes, vector clock size must be at least n in order to characterize the causality of events [4]. Singhal and Kshemkalyani also presented an efficient implementation of vector clocks [24]. Applications of vector clocks have been found in the field of distributed debugging [18, 12], especially in detecting global predicates [7, 11, 5, 10, 6]. Vector clocks have also been used in developing protocols which guarantee causal ordering among messages [2, 23], rollback recovery [19], and in many other distributed applications [20].

When using vector clocks, it is difficult to determine the optimal number of bits used to implement the integer variables for clock values. If the number is too small, overflow is imminent; if the number is too large, the extra cost of storing and maintaining clocks becomes intolerable. One straightforward strategy to relieve us from such a dilemma is to roughly estimate the necessary number of bits, and to reset vector clocks at each process when any of them is about to overflow. The purpose of this paper is to establish the necessary and sufficient condition for a correct clock resetting such that the functionality of vector clocks can be preserved. Our result indicates that for some applications, the potential clock overflow can be completely prevented by carefully choosing the condition for initiating the clock reset protocol.

Rest of this paper is organized as follows. Section 2 introduces our system model. Section

3 presents and proves a necessary and sufficient rule for a correct clock resetting. The implementation details of this rule is given in Section 4, with its applicability and limitation discussed in Section 5. Section 6 concludes this paper.

2 The Model

Three types of events occur in distributed systems: the *sending* of messages, the *receipt* of messages, and internal events [18]. The *happened-before* relation (denoted by $\rightarrow$) on the set of events is the smallest transitive relation satisfying the following conditions [15]: (1) if events a and b occur in the same process and if a comes before b then $a \rightarrow b$; (2) if a is a sending of a message m and b is the receipt of m, then $a \rightarrow b$. Events a and b are said to be *causally related* if $a \rightarrow b$ or $b \rightarrow a$; otherwise, a and b are said to be *concurrent*.

Let n be the number of processes in a distributed system. Each process P_i has its own vector clock $\mathcal{V}_i$ which is defined as an n-element integer vector. Every event e occurring in P_i is assigned a *timestamp* $\mathcal{T}(e)$, which is the content of $\mathcal{V}_i$ at the instant that e occurs. The rule by which each process evolves its clock and timestamps its events is as follows. (1) When an internal or sending event e occurs at process P_i, $\mathcal{V}_i[i]$ is increased by one, and then assigned to $\mathcal{T}(e)$. (2) Every message m is attached a timestamp $\mathcal{T}(m)$ equal to the timestamp of the event that sends m. When message m is received by process P_i, $\mathcal{V}_i[i]$ is incremented by one, and for all $k \neq i$, $\mathcal{V}_i[k] := \mathbf{max}(\mathcal{V}_i[k], \mathcal{T}(m)[k])$. The timestamp of this receipt event is then set to be the result $\mathcal{V}_i$.

Let $\mathcal{T}_i$ and $\mathcal{T}_j$ represent two distinct timestamps, between which an ordering relation *less-than* ($<$) is defined [24] as $\mathcal{T}_i < \mathcal{T}_j$ if $\mathcal{T}_i \neq \mathcal{T}_j \wedge \forall k : \mathcal{T}_i[k] \leq \mathcal{T}_j[k]$. The set of all events possesses the *vector clock property* [18]. That is, for any two events a and b, $a \rightarrow b \Longleftrightarrow \mathcal{T}(a) < \mathcal{T}(b)$. Note both the *happened-before* and the *less-than* relation are *precedence* [17] relations, which means that they are antisymmetric and transitive. Many researchers have extended these relations to include the reflexivity property, so that the terminologies and results pertaining

to *partial ordering* relations can be applied.

We call the action that a process takes to reset its vector clock a *reset event*. A collection of reset events, one from each process, constitutes a *reset cut*. The setup of a new reset cut concludes the current timestamping phase and starts a new one. Let us begin with Phase 1 and let E^i be the set of all events occurring in Phase i. It is necessary that the vector clock property holds for each E^i:

$$\forall i : \forall a, b \in E^i :: a \rightarrow b \Longleftrightarrow \mathcal{T}(a) < \mathcal{T}(b).$$

This is referred to as the restricted vector clock property (**RVCP**). Note that a reset event does not belong to any E^i.

Let *sent*(m) and *recv*(m) represent the sending and receipt of a message m, respectively. Suppose that *sent*$(m) \in E^i$ and *recv*$(m) \in E^j$. We call m a *normal* message if $i = j$, a *forward* message if $i < j$, or a *backward* message if $i > j$.

3 The Reset Rule

3.1 The Reset Rule is Necessary for RVCP

We define a *reset line* to be a temporal line connecting two reset events in the time-space diagram (refer to the dotted lines shown in Figure 1). A reset cut partitions the progression of clocks into two separate parts. However, a forward or a backward message which crosses some reset line can break the vector clock property. The delivery of a forward message propagates obsolete information about the sender's clock to the receiver, causing the receiver to incorrectly update its clock. So it is possible that there exist two events e_i and e_j in the same timestamping phase and that $\mathcal{T}(e_i) < \mathcal{T}(e_j)$ but $e_i \not\rightarrow e_j$. As an example, consider the time-space diagram shown in Figure 1. Suppose that each process P_i resets its respective vector clock at time t_i. The delivery of m_1 causes the timestamp of event b, which is the receipt of m_1, to leap to $[7, 2, 7]$. We can see that $\mathcal{T}(a) < \mathcal{T}(b)$ but $a \not\rightarrow b$. The delivery of a

backward message can also break RVCP. Message m_2 in Figure 1 is a such example. We can see that $c \rightarrow d$ but $\mathcal{T}(c) \not< \mathcal{T}(d)$. This problem arises since the clock evolution contributed by the reception of a backward message will vanish when later this clock is reset. Evidently, it is necessary to preclude the possibility of forward or backward messages to guarantee RVCP. This is explicitly stated as follows.

Reset Rule: Messages must not cross any reset line.

- All messages sent from process P_i to process P_j *before* P_i resets its clock must be received *before* P_j has reset its clock, thus precluding the possibility of forward messages.
- All messages sent from process P_i to process P_j *after* P_i resets its clock must be received *after* P_j has reset its clock, thus precluding the possibility of backward messages.

This rule demands synchronization between clock resetting and message receptions. In the next subsection we shall justify that the reset rule is not only necessary, but also sufficient to ensure RVCP.

3.2 The Reset Rule is Sufficient for RVCP

A set of events $\mathcal{E}$, together with the happened-before relation $\rightarrow$ on $\mathcal{E}$, constitutes an event structure $(\mathcal{E}, \rightarrow)$ that represents a distributed computation. A *causal chain* defined on $(\mathcal{E}, \rightarrow)$ is a sequence of events $e_1, e_2, \ldots, e_r$, where $r \geq 2$, such that $e_i \in \mathcal{E}, 1 \leq i \leq r$, and $e_1 \rightarrow e_2, e_2 \rightarrow e_3, \cdots, e_{r-1} \rightarrow e_r$. The set of all possible causal chains starting at event a and ending with event b is denoted by $\Phi(a, b)$. Let $\Sigma(\xi)$ denote the set of all elements contained in a causal chain ξ. A causal chain $\xi : e_1, e_2, \ldots, e_r$ is said to be a *closure* of $\Phi(e_1, e_r)$ if and only if $\forall i \in \{1, \ldots, r-1\} : \forall e' \in \mathcal{E} - \Sigma(\xi) :: e_i \not\rightarrow e' \vee e' \not\rightarrow e_{i+1}$.

Lemma 1 Given a finite event set $\mathcal{E}$ and two events $a, b \in \mathcal{E}$, each non-closure causal chain $\xi \in \Phi(a, b)$ can be extended to a closure of $\Phi(a, b)$.

Proof: Let $\xi : e_1, e_2, \ldots, e_r$ be a non-closure causal chain, where $a = e_1$ and $b = e_r$. It follows that $\exists i \in \{1, \ldots, r-1\} : \exists e' \in \mathcal{E} - \Sigma(\xi) :: e_i \rightarrow e' \wedge e' \rightarrow e_{i+1}$. By placing e' between e_i and e_{i+1}, we can form a new causal chain $\xi' : e_1, e_2, \ldots, e_i, e', e_{i+1}, \ldots, e_r$. If ξ' is a closure of $\Phi(a, b)$, then we are done. Otherwise, we follow the above argument and yield another causal chain. Since the number of events in $\mathcal{E}$ is finite, eventually we will obtain a a closure of $\Phi(a, b)$. $\Box$

Let $\phi(a, b) \subseteq \Phi(a, b)$ be the set of all closures of $\Phi(a, b)$. It turns out that for any two events a and b, $a \rightarrow b$ if and only if $\phi(a, b)$ is not empty. A causal chain $\xi : e_1, e_2, \ldots, e_r$, where $r \geq 2$, such that $\forall i \in \{1, \ldots, r-1\} : \mathcal{T}(e_i) < \mathcal{T}(e_{i+1})$ is said to be *monotonic*. Obviously, a closure causal chain whose elements are all in the same timestamping phase is monotonic. Thus we have the following invariant.

$$\forall a, b \in E^i : \exists \xi \in \phi(a, b) \wedge \Sigma(\xi) \subseteq E^i \Longrightarrow \mathcal{T}(a) < \mathcal{T}(b). \tag{1}$$

Theorem 1 The reset rule suffices to guarantee RVCP.

Proof: Suppose that RVCP does not hold. We shall prove that the reset rule is violated.

Case 1 $\exists a, b \in E^i : a \rightarrow b \wedge \mathcal{T}(a) \not< \mathcal{T}(b)$.

The relation $a \rightarrow b$ implies that $\phi(a, b)$ is not empty, while by (1), $\mathcal{T}(a) \not< \mathcal{T}(b)$ implies that there exists no causal chain ξ in $\phi(a, b)$ such that $\Sigma(\xi) \subseteq E^i$. Therefore, for each causal chain $\xi : e_1, e_2, \ldots, e_r$ in $\phi(a, b)$, where $a = e_1$, $b = e_r$, and $r > 2$, we can find the minimal j, $2 \leq j \leq r-1$, such that $e_j \in E^{j'}$ and $j' < i$, or we can find the maximal k, $2 \leq k \leq r-1$, such that $e_k \in E^{k'}$ and $k' > i$. The former case indicates a backward message sent by e_{j-1} to e_j, while the latter indicates the presence of a backward message from e_k to e_{k+1}.

Case 2 $\exists a, b \in E^i : \mathcal{T}(a) < \mathcal{T}(b) \wedge a \not\rightarrow b$.

Since timestamps always take monotonically increasing values in the same timestamping phase, this case arises only if the timestamp of event b is illogically enlarged. The

only way for processes to illogically enlarge vector contents is to receive forward messages whose timestamps are obsolete but larger than those of the receivers. Message m_1 in Figure 1 is a such example.

□

4 The Implementation of the Reset Rule

Theorem 1 implies that a correct reset cut must be a *strongly consistent* cut [9, 13], i.e., a consistent cut without in-transit messages (forward messages in terms of our definition). In the following we present a coordinating protocol that yields on-the-fly reset cuts.

4.1 The Algorithm

Our approach is inspired by Chandy and Lamport's distributed snapshot algorithm [3]. We assume that between any two processes, there is at most one communication channel connected which provides bidirectional, reliable, and FIFO-ordered delivery. Message transmission delays are assumed to be arbitrary but finite.

Two kinds of control messages are used by our protocol: *reset_req* and *reset_done*. Each process P_i, which can operate in one of the three modes *normal*, *mute*, and *stand-by*, maintains a variable S_i that records P_i's current process mode. Let $\eta(P_i)$ denote the set of processes having a communication channel connected to P_i. For each of P_i's neighbor, $P_j \in \eta(P_i)$, P_i maintains a variable, $S_{i,j}$, that records P_j's process mode currently known by P_i. S_i and $S_{i,j}$ are initiated to be *normal* for all i, j.

The execution of our algorithm is triggered by some condition local to a process (which will be discussed later). A process that starts the execution is called an initiator, which first sends control message *reset_req* to each of its neighbors and then enters *mute* mode. A process operating in *mute* mode is not allowed to send application messages. Any non-

initiator process P_i operating in *normal* mode, on receiving *reset_req* for the first time, behaves like an initiator. That is, it sends out *reset_req* message to each of its neighbors, and then enters *mute* mode. In the mean time, it sets $S_{i,k}$ to *mute*, if P_k is the process that sent *reset_req* to P_i. When P_i in *mute* mode receives a control message from one of its neighbors, say, P_j, it sets $S_{i,j}$ according to the message it receives: it sets $S_{i,j}$ to *mute* on receiving *reset_req*, and sets $S_{i,j}$ to *stand-by* on receiving *reset_done*.

At P_i, when the values of all $S_{i,j}$'s have been changed from *normal* to *mute* or *stand-by*, P_i resets its clock, sends control messages *reset_done* to all its neighbors, and then enters *stand-by* mode. A process operating in *stand-by* mode can send application messages to another only if the former has recorded the latter's mode as *stand-by*. Finally, after the values of all $S_{i,j}$'s have been changed to *stand-by*, P_i sets all $S_{i,j}$'s to *normal*, and enters back to *normal* mode, which indicates the completion of the current run of the protocol on this process. The detailed algorithm of the clock reset protocol executing in P_i is shown in Figure 2. Note both message-driven routines are atomic, i.e., non-interruptable during its execution.

4.2 Correctness Justification

We now justify that the presented protocol correctly implements the clock reset rule. For any two adjacent processes P_i and P_j, P_j does not send any application message to P_i after it sends *reset_req* to P_i and before it resets $\mathcal{V}_j$. Because message delivery is FIFO, the delivery of P_j's *reset_req* message on P_i indicates that all application messages sent by P_j before $\mathcal{V}_j$ is reset have been received. Since P_i resets its clock after it has received *reset_req* along each incoming channel, our protocol precludes the possibility of forward messages.

After resetting $\mathcal{V}_i$, P_i sends application messages to P_j only after it has received *reset_done* from P_j, which indicates that P_j has already reset its own clock. Therefore our protocol also precludes the possibility of backward message.

The process that initiates the reset protocol effectively plays the role of a diffusing source of *reset_req* messages. If two or more processes initiates the protocol simultaneously, there will be multiple sources that spread *reset_req* messages. Concurrent initiations of the protocol does not cause any correctness problem, since the correctness of the protocol relies on the fact that eventually *reset_req* messages are spread over every communication channel, no matter how many diffusing sources there will be.

4.3 Eliminating *Reset_done* Messages

Control message *reset_done* is used to prevent the occurrence of backward messages. It can be eliminated if we modify the original protocol as follows. (1) Each process now operates in two possible modes: *normal* or *mute*. (2) As soon as P_i resets its clock, it is allowed to send out application messages to any adjacent process. (3) When P_i operating in *mute* mode receives an application message m sent from P_j, it examines the value of $S_{i,j}$ to decide what action should be taken. If the value of $S_{i,j}$ is *normal*, m must have been sent before P_j resets $\mathcal{V}_j$ and therefore can be accepted; if $S_{i,j} = mute$, m must have been sent after P_j resets $\mathcal{V}_j$ and thus needs to be buffered. The buffered messages will not be accepted or processed until P_i resets $\mathcal{V}_i$.

With this modification, potential backward messages are not inhibited by their sender. Instead, they are buffered at the receiver site and will not be processed until the receiver has reset its clock. So the correctness of the original protocol is still preserved.

This approach reduces message cost, but we need to pay for storage cost instead. If available storage for buffering backward messages at some process is limited, all other processes sending messages to this process must be careful not to overrun its buffer. Suppose that P_j has a storage buffer which is capable of storing r messages for P_i. After P_i resets $\mathcal{V}_i$ but before any application message is received from P_j, P_i has no way to tell if P_j has reset $\mathcal{V}_j$ or not, so the maximal number of messages allowed to be sent from P_i to P_j is limited to r. P_i has to suspend sending to P_j after it has sent out r application messages to P_j, unless

and until it has received an application message from P_j, which indicates the completion of $\mathcal{V}_j$'s reset.

5 Discussion

5.1 The Triggering Condition of the Protocol

The triggering condition of the reset protocol must be appropriately set up so that vector clocks do not overflow before being reset. Mattern [18] showed that at any instant of time, $\forall i, j : \mathcal{V}_i[i] \geq \mathcal{V}_i[j]$. Therefore, if each process P_i can ensure that clock entry $\mathcal{V}_i[i]$ will not get overwhelmed, overflow is not possible. Generally, $\mathcal{V}_i[i]$ is incremented by one every time a message is sent, a message is received, or an internal event occurs. In some applications, however, we do not concern for the causality of internal events, and vector clocks do not advance on the occurrence of internal events. We can prevent clock overflow in this kind of applications by constraining the number of messages allowed to be sent within each timestamping phase. Specifically, each process counts the number of messages it sends in a per-channel basis, and initiates the reset protocol when some of its counting value corresponding to a particular channel reaches a predefined limit for that channel. Since if all processes constrain the number of messages they send, the number of messages they may receive is also bounded, there is no need to also constrain the number of messages a process is allowed to receive.

Let $L_{i,j}$ denote the maximal number of messages allowed to be sent from P_i to P_j. With our reset protocol, an overflow-free setting of $L_{i,j}$ must satisfy the following inequality:

$$\forall i : \sum_{P_j \in \eta(P_i)} (L_{i,j} + L_{j,i}) \leq t_i, \tag{2}$$

where t_i denotes the maximal value of $\mathcal{V}_i[i]$. Finding a triggering condition subject to (2) is not difficult. As an example, $L_{i,j}$ can be set to

$$\frac{1}{2} \times \mathbf{min}(\lfloor \frac{t_i}{|\eta(P_i)|} \rfloor, \lfloor \frac{t_j}{|\eta(P_j)|} \rfloor)$$

for each P_i and $P_j \in \eta(P_i)$. However, finding a triggering condition both to satisfy (2) and to constrain message sending only when necessary is impossible without prior knowledge of run-time behavior of the processes in the system.

5.2 The Insufficiency of RVCP

For some applications, we may have to compare timestamps of events occurring in different timestamping phases. RVCP does not help in this case, thus we need an auxiliary function for this kind of event comparisons. Let a and b be two events occurring respectively in timestamping phases E^i and E^j $(i < j)$. It is impossible that b happens before a, since the reset protocol precludes the possibility of any backward messages. The auxiliary function therefore only needs to decide whether a happens before b or a and b are concurrent. Figure 3 shows the auxiliary function which attempts to find a causal chain form a to b by referring to a set of intermediate events, specifically, the set of each process's last event in each timestamping phase between E^i and E^{j-1}. This function returns *true* if $a \rightarrow b$ and *false* otherwise.

Unfortunately, implementing such a rule will inevitably involve attaching each event an additional variable indicating the number of the current timestamping phase, which has essentially the same unfavorable effect as adding an extra entry in the clock vector. Moreover, the variable storing the timestamping phase number may also overflow. How this problem can be dealt with depends on available domain knowledge about the applications. For example, if a vector-clock application never needs to examine causal relationship between events that are two or more timestamping phases apart, a three-valued counter is sufficient to represent the current timestamping phase number. Let $\{1, 2, 3\}$ be the set of possible values. We can explicitly define that E^1 precedes E^2, E^2 precedes E^3, and E^3 precedes E^1. These three values thus can be cyclicly used without worrying about overflow.

5.3 Applicability of the Protocol

In the application of preserving causal message ordering [2, 21, 23], vector clocks advance only when a message is sent or received, so clock overflow can be completely prevented by setting up a triggering condition satisfying (2). Vector clocks in this kind of applications are mainly used to timestamp messages. Timestamps are to be examined by receiver processes to determine if received messages can be delivered. Once a message has been delivered, its timestamp becomes useless and can be safely discarded. Since our protocol flushes all messages between timestamping phases, there is no possibility to compare two timestamps or vector clock contents that are from different timestamping phases. Therefore, the insufficiency of RVCP does not cause a problem.

In some applications, vector clocks advance only when a message is sent or received, and are used to timestamp events or states as well as messages. However, only certain types of events or states are of interest, and so are their timestamps. For example, in applications that exploit vector clock to detect global predicates [11, 5, 10, 6], only local states that satisfy some particular local predicate are of interest. Their timestamps are locally collected and on which a consistent global predicate is to be identified by either a centralized checker process or a set of cooperative processes. Since this kind of clocks does not advance on the occurrences of internal events, clock overflow will not happen if we set up trigger conditions satisfying (2). However, the insufficiency of RVCP does pose a problem, as noted in Section 5.2, in identifying consistent global states. We believe that this problem has no satisfactory solution other than using extra counter bits to represent phase numbers.

In other applications, clocks advance even on the occurrences of internal events, and it is needed to examine timestamps that are from different timestamping phases. The insufficiency of RVCP remains to be an inherent problem. Since internal events may occur arbitrarily, it is impossible to prevent clock overflow unless we can suspend a process's computation during the execution of the reset protocol. Suspending a process's computation is usually considered unacceptable. Therefore, for this kind of applications, clock resetting is

not appropriate.

5.4 Related Work

Our necessary and sufficient condition for clock reset can be related to the consistent snapshot recording problem [3] in the sense that a correct reset cut forms a consistent snapshot with no in-transit messages, if each reset event is viewed as an event that takes local snapshot. However, existing solutions to this problem [3, 14, 16, 13, 1] do not preclude the existence of in-transit messages, and thus cannot be adopted as a reset protocol. Fischer et al. [9] proposed a method of taking strongly consistent (i.e., no in-transit message) global checkpointings for distributed transaction system. Their method is suitable only for an off-line analysis of the entire system, and thus cannot be used to produce on-the-fly reset cuts.

Birman et al. proposed a flushing protocol which is used to cope with the changes of group membership in a process group [2]. After executing the flush protocol, all processes can reset their clocks, so clock overflow can be prevented in some way. Our method is similar to their timestamp reinitializing technique. Both approaches use two-phase flushing protocol to conclude a timestamping phase and start the next one. Additionally, both require that message sending should be inhibited during the execution of the flushing protocol.

However, Birman's method resets vector clocks after the flush protocol is completed, while ours does so as soon as the first phase of the flushing protocol has been completed. Moreover, in our protocol, after resetting its clock a process can start communicating with another process provided that the former has been informed of the latter's reset action (as explained in Section 4.1). As a consequence, the time period during which message sending is inhibited will be much shorter in our protocol.

6 Concluding Remarks

For many vector-clock applications, our scheme relieves us from the difficult task of determining the optimal number of bits to implement vector clocks. One only has to determine a triggering condition for the reset protocol such that the overhead of strongly consistency enforcement between phases can be tolerated while clock overflow can be prevented.

Although we are primarily concerned with vector clock reset, the established result can also applied to matrix clocks [20, 22].

References

[1] Acharya, A., and Badrinath, B. R. Recording distributed snapshots based on causal order of message delivery. *Inform. Process. Lett.* **44**, Dec. 1992, 317–321.

[2] Birman, K., Schiper, A., and Stephenson, P. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.* **9**, 3 (1991), 272–314.

[3] Chandy, K. M., and Lamport, L. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.* **3**, 1 (Feb. 1985), 63–75.

[4] Charron-Bost, B. Concerning the size of logical clocks in distributed systems. *Inform. Process. Lett.* **39**, 1 (1991), 11–16.

[5] Chiou, H.-K., and Korfhage, W. Efficient global event predicate detection. *Proc. 14th International Conference on Distributed Computing Systems*, June 1994, pp. 642–649.

[6] Chiou, H.-K., and Korfhage, W. Enhancing distributed event predicate detection algorithms. *IEEE Trans. Parallel and Distrib. Syst.* **7**, 7 (July 1996), 673–676.

[7] Cooper, R., and Marzullo, K. Consistent detection of global predicates. *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices* **26**, 12 (Dec. 1991), 167–174.

[8] Fidge, J. Timestamps in message-passing systems that preserve the partial ordering. *Proc. 11th Australian Computer Science Conference*, Feb. 1988, pp. 56–66.

[9] Fischer, M. J., Griffeth, N. D., and Lynch, N. A. Global states of a distributed system. *IEEE Trans. Software Engrg.* **SE-8**, 3 (May 1982), 198–202.

[10] Garg, V. K., and Chase, C. M. Distributed algorithms for detecting conjunctive predicates. *Proc. 15th International Conference on Distributed Computing Systems*, June 1995, pp. 423–430.

[11] Garg, V. K., and Waldecker, B. Detection of weak unstable predicates in distributed programs. *IEEE Trans. Parallel and Distrib. Syst.* **5**, 1 (Mar. 1994), 299–307.

[12] Goldberg, A. P., Gopal, A., Lowry, A., and Strom, R. Restoring consistent global states of distributed computations. *ACM SIGPLAN Notices* **26**, 12 (Dec. 1991), 144–154.

[13] Helary, J.-M. Observing global states of asynchronous distributed applications. *Proc. Third International Workshop on Distributed Algorithms*, 1989, pp. 124–135.

[14] Lai, T. H., and Yang, T. H. On distributed snapshots. *Inform. Process. Lett.* **25**, May 1987, 153–158.

[15] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM* **21**, 7 (July 1978), 558–565.

[16] Li, H. F., Radhakrishnan, T., and Venkatesh, K. Global state detection in non-FIFO networks. *Proc. 7th International Conference on Distributed Computing Systems*, Sep. 1987, pp. 364–370.

[17] Liu, C. L. *Elements of Discrete Mathematics*. McGraw-Hill, 1985, page 119.

[18] Mattern, F. Virtual time and global states of distributed systems. In M. C. et al. (Eds.). *Proc. International Workshop on Parallel and Distributed Algorithms*. North-Holland, Elsevier Science, 1989, pp. 215–226.

[19] Peterson, S. L., and Kearns, P. Rollback based on vector time. *Proc. 12th Symposium on Reliable Distributed Systems*, Oct. 1993, pp. 68–77.

[20] Raynal, M. About logical clocks for distributed systems. *Oper. Syst. Rev.* **26**, 1 (Jan. 1992), 41–48.

[21] Raynal, M., Schiper, A., and Toueg, S. The causal ordering abstraction and a simple way to implement it. *Inform. Process. Lett.* **39**, 6 (Sep. 1991), 343–350.

[22] Raynal, M., and Singhal, M. Logical time: capturing causality in distributed systems. *IEEE Computer* **30**, 2 (Feb. 1996), 49–56.

[23] Schiper, A., Eggli, J., and Sandoz, A. A new algorithm to implement causal ordering. *Proc. Third International Workshop on Distributed Algorithms*, 1989.

[24] Singhal, M., and Kshemkalyani, A. An efficient implementation of vector clocks. *Inform. Process. Lett.* **43**, 1 (1992), 47–52.

Author Biographies

Li-Hsing Yen received the B.S. and M.S. degrees in computer science and information engineering, both from National Chiao Tung University, Hsinchu, Taiwan, in 1989 and 1991, respectively. Since September 1993, he has been a Ph.D. student in the Department of Computer Science and Information Engineering at National Chiao Tung University, Hsinchu, Taiwan. His current research interests include distributed algorithms, program testing and verification, and mobile computing. E-mail: lsyen@csie.nctu.edu.tw.

Captions

Fig. 1: A reset cut with a forward message m_1 and a backward message m_2

Fig. 2: The clock reset protocol in process P_i

Fig. 3: A function implementing the auxiliary rule.

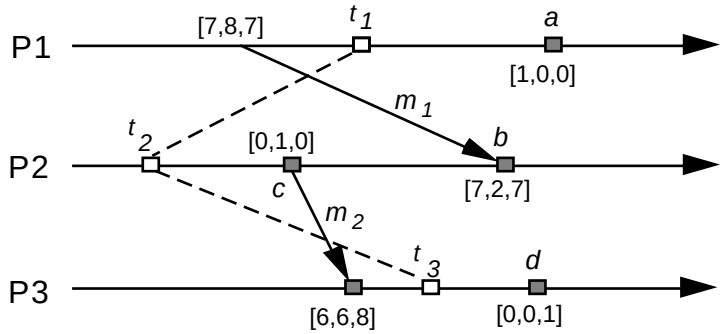

Figure 1:

Upon receiving a *reset_req* **message from** P_j **do**
    if $S_i = normal$ **then** /* receives *reset_req* for the first time */
        $S_i \leftarrow mute$ /* enters *mute* mode */
        send *reset_req* to all $P_k \in \eta(P_i)$
    end if
    $S_{i,j} \leftarrow mute$ /* records P_j's mode as *mute* */
    if $S_{i,k} \neq normal$ for all $P_k \in \eta(P_i)$ **then**
        reset P_i's clock
        send *reset_done* to all $P_k \in \eta(P_i)$
        $S_i \leftarrow$ *stand-by* /* enters *stand-by* mode */
    end if
end.

Upon receiving a *reset_done* **message from** P_j **do**
    $S_{i,j} \leftarrow$ *stand-by* /* records P_j's mode as *stand-by* */
    if $S_i =$ *stand-by* and $S_{i,k} =$ *stand-by* for all $P_k \in \eta(P_i)$ **then**
        set $S_{i,k}$ to *normal* for all $S_{i,k}$
        $S_i \leftarrow normal$ /* enters *normal* mode */
    end if
end.

Figure 2:

Function *happen-before*(a, b)

/* Assume that a and b occur in E^i and E^j respectively $(i < j)$ */

/* Let $\odot$ represent the bit-wise "**and**" operator taken in pair-wise manner */

Let $\mathcal{L}_k^p$ represent the timestamp of P_k's last event in E^p

Let IV be an integer vector of length n, initially $[0, 0, \ldots, 0]$.

for $l = 1$ to n **do**

  if $\mathcal{T}(a) \leq \mathcal{L}_l^i$ **then** $IV(l) \leftarrow 1$

end for

for $k = i + 1$ to $j - 1$ **do**

  for $l = 1$ to n **do**

    if $\mathcal{L}_l^k \odot IV \neq [0, 0, \ldots, 0]$ **then** $IV(l) \leftarrow 1$

  end for

end for

if $\mathcal{T}(b) \odot IV \neq [0, 0, \ldots, 0]$ **then** return *true*

else return *false*

end.

Figure 3: