ISSN:0975-9646

Matrix Clock Synchronization in the Distributed Computing Environment

Avaneesh Singh[#1], Neelendra Badal[#2]
[#1#2]*Computer science and Engg. Department at Kamla Nehru Institute of Technology, Sultanpur, U.P., India*

Abstract- **Matrix clock is a generalisation of the notion of vector clock. Matrix clock is a mechanism for capturing chronological and causal relationship in a Distributed system. Matrix clock is a list of vector clocks, and it also contains the current state of each node in the system. On the basis of this it know which peer received already which messages. When it exchanges messages with another node in the system then it compare the matrix clocks and remember always the highest values for each node. After retrieve back information of node it delete messages which were sent before, it considered that the node already have received them. Matrix clock allows establishing a lower bound on what other hosts know, and is useful in applications such as check pointing and garbage collection.**

Keywords- **Distributed computing, Vector clock, Matrix clock, Event ordering, Clock synchronization, Logical clock.**

I. Introduction

A matrix clock is an extension of the vector clocks that also contains the information about the other processes views of the system. **Matrix clock** is key to the solution of above problem is that the send and receive events corresponding to a particular message-passing action can be unambiguously matched. Any other ways of retaining this information, such as annotating send and receive timestamps with a unique message identifier at single matrix called **matrix clock** implementation, will be effective. In a landmark article, Lamport [1] defined the causal relationships among events occurring in a message passing distributed computation as the smallest relation → such that

(i) if e and f are events in the same process, and e occurs before f , then $e \rightarrow f$,
(ii) if event e denotes transmission of a message m by a process, and event f denotes reception of the same message m by another process, then $e \rightarrow f$, and
(iii) if $e \rightarrow f$ and $f \rightarrow g$, then $e \rightarrow g$.

II. Vector Clock

A number of researchers, most notably Mattern [2] and Fidge [3], later independently proposed *vector clocks* as a timestamping mechanism for distributed computations that captures causality.

In a computation involving n parallel processes, each process p maintains a logical clock vector of length n×n. These vectors are used to timestamp each event e and are also piggybacked onto each outgoing message m. Let $\vec{p}$,$\vec{e}$ and $\vec{m}$ be the vectors associated with the respective process clock, event timestamp and piggybacked message matrices [4].

For some vector $\vec{v}$ let $\vec{v}$(i) denotes its ith element. Vector elements act as counters of the number of events known to have occurred in each process. They are maintained using the following steps.

i. For each process p, all elements of $\vec{p}$ are initially 0.
ii. When process p performs some *internal* event e, it
   a. Increments process clock element $\vec{p}$(p), and
   b. Sets event timestamp $\vec{e}$ equal to $\vec{p}$.
iii. When process p performs a *send* event e, that produces a message m,
   a. it increments process clock element $\vec{p}$(p),
   b. Sets event timestamp $\vec{e}$ equal to $\vec{p}$, and
   c. Sets the piggybacked timestamp $\vec{m}$ attached to the outgoing message equal to $\vec{p}$.
iv. When process p performs a *receive* event e, that accepts a message m with piggybacked timestamp $\vec{m}$, it
   a. Increments process clock element $\vec{p}(p)$,
   b. Sets each process clock element $\vec{p}$(i) equal to max $(\vec{p}(i), \vec{m}(i))$, where i ranges from 1 to n, and
   c. Sets event timestamp $\vec{e}$ equal to $\vec{p}$.

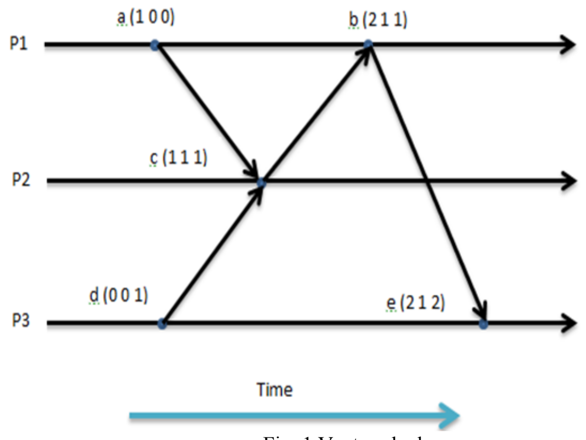

Fig. 1 Vector clock

III. Limitations Of Vector Clock

It is tempting, therefore, to conclude that a set of vector timestamps, one per event, fully characterize a distributed computation. However, we observe that in systems that allow message ‘overtaking’ this is not necessarily so. Fig. 2 shows two distinct computations that have identical event timestamps. On the left events a and d are internal to processes P_1 and P_2. However, in the computation on the right event *a* is a send, and event *d* is a receive. Despite the obvious differences between the computations, the rules for maintaining vector clocks in Section 2 timestamp their events identically. An attempt to reconstruct either of these computations accurately from the event timestamps alone would be thwarted by this ambiguity [5].

Thus, merely prohibiting overtaking of messages between *pair* of processes is not sufficient to avoid the problem. One may think that keeping track of the 'type' of events, i.e., whether they are internal, sends or receives, would resolve the problem [7]. Certainly this information would be sufficient to disambiguate the computations in Fig. 2. Corresponding events in the computations receive the same timestamps, and have the same 'types', but the computations are different [6].

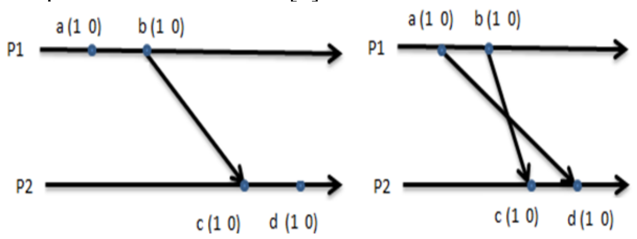

Fig. 2 Identical timestamp within two different computations

Solution

Matrix clock is key to the solution of above problem is that the send and receive events corresponding to a particular message-passing action can be unambiguously matched. Any other ways of retaining this information, such as annotating send and receive timestamps with a unique message identifier at single matrix called **matrix clock** implementation, will be effective.

IV. Matrix Clock

A. Definition

Matrix clocks is an extension of the vector clocks that also contains the information about the other processes views of the system.

In a system of matrix clocks, the time is represented by a set of n × n matrices of non-negative integers. A process p_i maintains a matrix mt_i [1...n, 1...n] where

- mt_i [i,i] denotes the local logical clocks of p_i and track the progress of the computation at process p_i.
- mt_i[i,j] represent the latest knowledge that process p_i has about the knowledge of p_i has about the logical local clock of p_j. The entire matrix mt_i denotes p_i's local view of the logical global time.

Note that row mt_i[i,.] is nothing but the vector clock vt_i[.] and exhibits all properties of vector clocks.

Processes p_i uses the following rules Rule R1 and Rule R2 uses to update its clock:

- Rule R1 : Before executing an event and it update its local logical time as follows: $mt_i[i,i] = mt_i[i,i]+d \quad (d > 0)$
- Rule R2: Each message m is piggy-backed with matrix time mt. When p_i receives a message (m, mt) from a process p_j, p_i executes the following sequence of actions:
  - o Update its logical global time as follows:
    $1 \le k \le n : mt_i [i,k] = \max(mt_i[i,k], mt[j,k])$
    $1 \le k , l \le n : mt_i [k,l] = \max(mt_i[k,l], mt[k,l])$
  - o Execute R1.
  - o Deliver message m.

Basic property:

Clearly vector mt_i [i,.] contains all the properties of the vector clocks. In addition Matrix clocks have the following property:

$\min(mt_i [k,l]) \ge t$, where process p_i knows that every other process p_k knows that p_i's local time has progressed till t

If this is true, it is clear that process p_i know that all other processes know that p_i will never send information with a local time $\le t$. In many applications, this implies that process will no longer require from p_i certain information and can use this fact to discard obsolete information.

If d is always 1 in the rule R1, then mt_i [k,l], denotes the number of events occurred at p_i and known at p_k as for as p_i's knowledge is concerned.

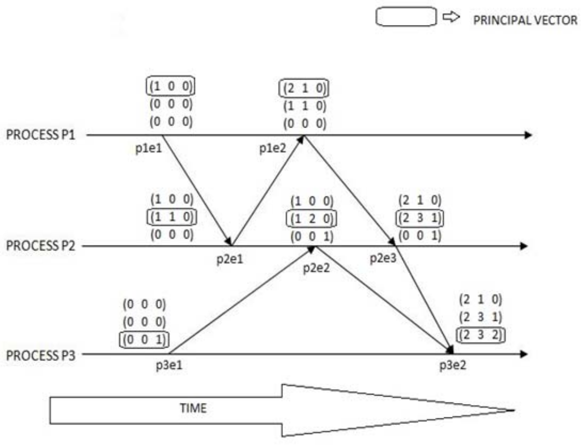

Fig. 3 The Matrix clock

B. Implementation of matrix clock:

(1)

(a) Initially all the matrices of events are zero for each process.

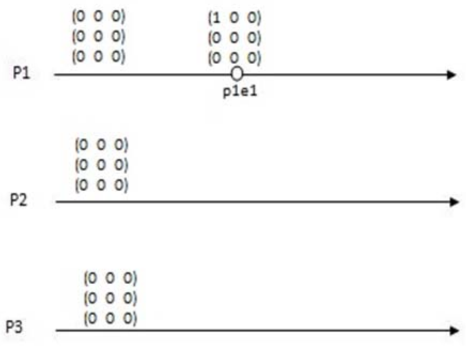

Fig. 4 Initial form of matrix clock

(b) If there is n processes run at same time then every event contain n × n matrix.

(c) Each event has n vector clocks, every one for each process

(d) The *n* th vector on process *i* is called process *n* 's principle vector

(e) Principle vector is the same as vector clock before

(f) Non-principle vectors are just piggybacked on messages to update "knowledge".

(2) Whenever any events occur then increment the clock value of vector.

(3) If we perform any internal event e in process then

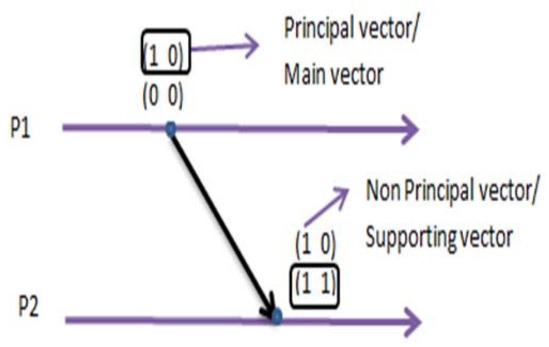

Fig. 5 Example of matrix time stamping

(a) In principal vector:

i. Increment the process clock element.

ii. Set the timestamp e equal to p.

(b) In non-principal vector :

i. Initially all the non-principal vectors are zero.

ii. If event come from any other process then at the position of non-principal vector put the principal vector where it come from.

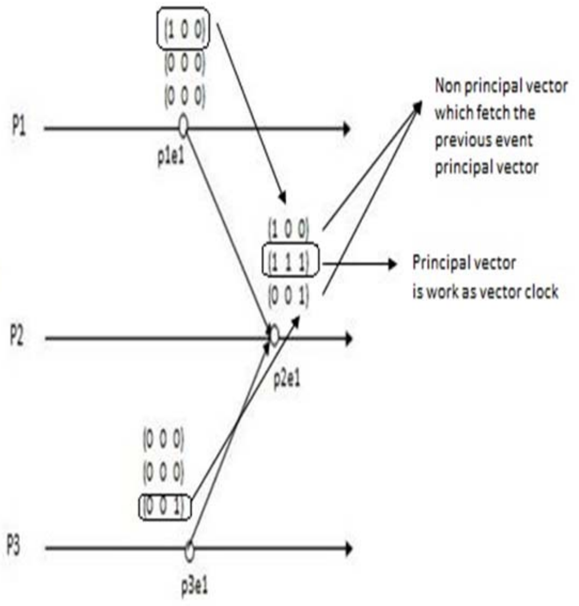
(1 0 0)
(0 0 0)
(0 0 0)
P1
p1e1
Non principal vector which fetch the previous event principal vector
(1 0 0)
(1 1 1)
(0 0 1)
Principal vector is work as vector clock
P2
p2e1
(0 0 0)
(0 0 0)
(0 0 1)
P3
p3e1

Fig. 6 How to work principal and non-principal vector in matrix clock

Note: Non principal vector are used to know about the other process information.

(4) When process perform the **send operation** (send event) e, that gives a message m, then it:

In principal vector:

i. After sending the event operation increments the process clock element p.

ii. Sets event timestamp e equal to p, and

iii. Sets the piggy-backed timestamp m attached to the outgoing message equal to p.

In non-principal vector:

i. Fetch the non-principal vectors from the previous related event's principal vector, for information.

ii. Previous related event's principal vector = present event's non principal vector.

(5) When processes perform the **receive operation** (receive messages) receive event e, that gives a message m with piggy-backed timestamp m , then it gives some following rules:

In principal vector:

i. After receiving the event increment process clock element p.

ii. Setting each process element p equal to Max (p (i), (m (i)), Where i range from 1 to n

iii. Set the event timestamp e equal to p.

In non-principal vector:

i. Fetch the previous events principal vectors and put is as non-principal vector.

Note:

(1) Principal vector as active side of matrix, which is important and it works as vector clock.

(2) Non principal vector are non-active side, but it store the important information of the previous principal vectors, which is very important to synchronization of event ordering of the messages.

V. EFFICIENCY OF TECHNIQUE

Now compute the efficiency of the proposed compaction technique. We define the efficiency of a compaction technique as the average percentage reduction in the size of the timestamp related information to be transferred in a message as compared to when sending the entire matrix timestamp. We also define the following term:

n: the average number of entries in T_i that qualify for transmission in a message using the proposed technique.

b: the number of bits in a sequence number.

$\log_2 N$: the number of bits needed to code N process ids.

The mattern / fidge clock require N.b bits of timestamp information, whereas the proposed technique require $(\log_2 N+b).n^2$ bits and vector timestamp technique require $(\log_2 N+b).n$ bits [8]. Thus the efficiency of the technique is given by the following expression:

$$\left\{1- \frac{(\log_2 N+b).n^2}{b.N}\right\}*100\ \%$$

It is easy to see that the propped technique is beneficial only if $n < N.b/(\log_2 N+b)$.

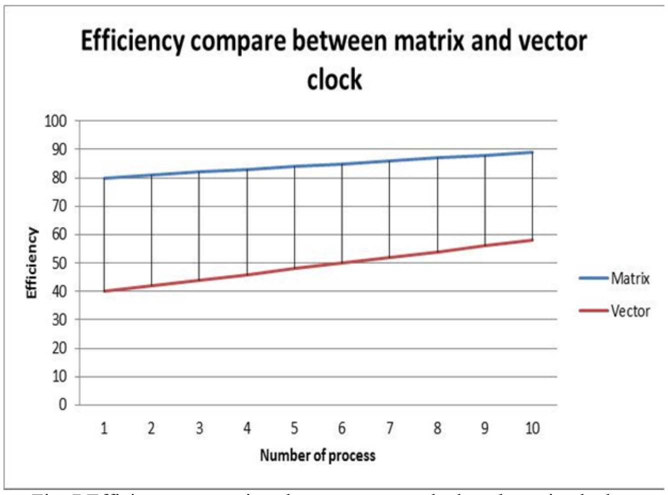

Fig. 7 Efficiency comparison between vector clock and matrix clock

TABLE 1 TIMESTAMP VALUE AT SINGLE EVENT

Timestamp value at single event			
Number of processes	Lamport clock	Vector clock	Matrix clock
1	1	1	1
2	1	2	4
3	1	3	9
4	1	4	16
5	1	5	25

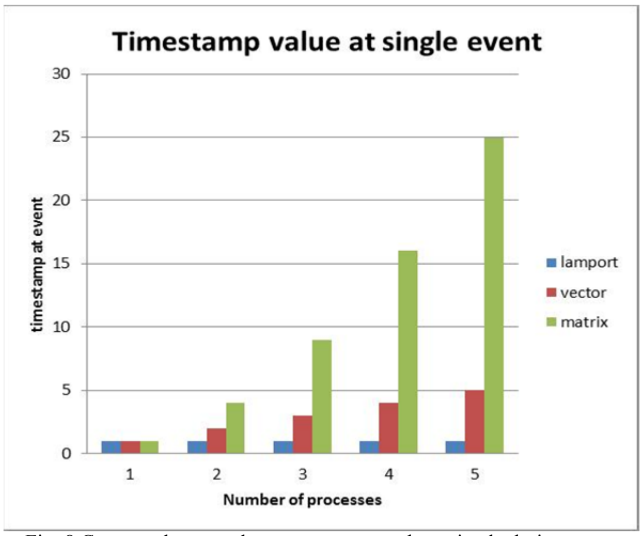

Fig. 8 Compare between lamport, vector, and matrix clock timestamp value at single event

VI. CONCLUSION

We have described about the matrix clock for characterizing distributed computations, and have examined its implementation and closure property In this paper we have presented an efficient technique to maintain the matrix clock, which cuts down the communication overhead due to propagation of matrix timestamp by sending only incremental changes in the timestamp. The technique can cut down the communication overhead substantially if the interaction between processes is localised. The technique has a small memory overhead. However this is not serious because main memory is cheap and is available in large quantities, and it is more desirable to reduce traffic on a communication network whose capacity is limited and is often the bottleneck.

ACKNOWLEDGMENT

I wish to thank the respected referees for pointing out a number of important references, and the professors of the Kamla Nehru Institute of Technology, Sultanpur in Distributed Computing course for helpful discussion and giving favourable environment.

REFERENCES

[1] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Comm. ACM 21 (7) (July 1978) 558–565.
[2] F. Mattern, Virtual time and global states of distributed systems, in: M. Cosnard et al. (Eds.), Parallel and Distributed Algorithms, North-Holland, Amsterdam, 1989.
[3] C.J. Fidge, Timestamps in message-passing systems that preserve the partial ordering, Australian Comput. Sci. Comm. 10 (1) (February 1988) 56–66.
[4] C.J. Fidge, Fundamentals of distributed system observation, IEEE Software 13 (6) (November 1996) 77–83.
[5] Raynal M. and Singhal M., "Logical time: Capturing causality in distributed systems," Computer, vol. 29, pp. 49-56, 1996.
[6] Fidge C., "Logical time in distributed computing systems," IEEE Computer, vol. 24, pp. 28-33, 1991.
[7] C.J. Fidge, A limitation of vector timestamps for reconstructing distributed computations, in: Elsevier Science, PII: S0020-0190(98) 0014 3- 4, 1998, *Information Processing Letters 87–91.*
[8] Mukesh Singhal, Ajay Kshemkalyani, " An efficient implementation of vector clocks", in Elsevier Science publishers, 0020-0190 /92 1992, *Information Processing Letters 90–92.*

AUTHORS

Avaneesh Singh received his Bachelor of Technology Degree in Computer Science and engineering in 2012 from Goel Institute of Technology and Management ,Lucknow, India, affiliated to Gautam Buddh Technical University (Formerly Uttar Pradesh Technical University) Lucknow, India. Currently he is pursuing Master of Technology in Computer Science and Engineering from Kamla Nehru Institute of Technology (An Academic Autonomous Govt. Engg. Institute), Sultanpur (U. P.), India, affiliated to Uttar Pradesh Technical University, Lucknow, India. His research areas of interests are Distributed Computing system.

Dr. Neelendra Badal is an Assistant Professor in the Department of Computer Science & Engineering at Kamla Nehru Institute of Technology, (KNIT), at Sultanpur (U.P.), INDIA. He received B.E. (1997) from Bundelkhand Institute of Technology (BIET), Jhansi (U.P.), INDIA, in Computer Science & Engineering, M.E. (2001) in Communication, Control and Networking from Madhav Institute of Technology and Science (MITS), Gwalior (M.P.), INDIA and PhD (2009) in Computer Science & Engineering from Motilal Nehru National Institute of Technology (MNNIT), Allahabad (U.P.), INDIA. He is Chartered Engineering (CE) from Institution of Engineers (IE), India. He is a Life Member of IE, IETE, ISTE, CSI, India. He has published more than 50 papers in International/National Journals, conferences and seminars. His research interests are Distributed System, Parallel Processing, GIS, Data Warehouse & Data mining, Software engineering and Networking