Information Processing Letters 43 (1992) 47–52
North-Holland

10 August 1992

An efficient implementation of vector clocks

Mukesh Singhal and Ajay Kshemkalyani

Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210, USA

Communicated by L. Kott
Received 4 January 1991
Revised 18 June 1991 and 28 January 1992

Abstract

Singhal, M. and A. Kshemkalyani, An efficient implementation of vector clocks, Information Processing Letters 43 (1992) 47–52.

The system of vector clocks is an essential tool for designing distributed algorithms and reasoning about them. We present an efficient implementation of vector clocks that reduces the size of timestamp related information to be transferred in a message. The implementation assumes FIFO message delivery and is resilient to changes in the topology of the distributed system.

Keywords: Distributed computing systems, efficient implementation, logical time, vector time, causal ordering

1. Introduction

We model a distributed system as a graph (V, E) with arbitrary connectivity, where V is the set of processes and $E \subseteq V \times V$ is a set of unidirectional logical FIFO channels. Process a can communicate with process b if there is a logical channel from a to b. Events at a process can be internal events, message send events, or message receive events. Lamport defines *happens before* (or the causality relation), an irreflexive, partial order on the set of events as follows [7]: For events e and e', e *happens before* e' is the smallest transitive relation satisfying the following conditions, (i) e and e' are events in the same process and e precedes e', and (ii) e is a message send event and e' is the corresponding message receive event. Lamport further defines a system of clocks wherein the "timestamp" of an event is the local clock value and e *happens before* $e' \Rightarrow$ *timestamp*$(e) <$ *timestamp*(e').

Recently, Mattern and Fidge formalized the notion of vector clocks which exactly characterize the causal order among events in a distributed system [5,10]. In this paper, we present an efficient technique to maintain vector clocks. In vector clocks, logical time is defined to be a vector of length N. (N is the number of processes in the system.) The logical time at process i is T_i and the timestamp of a message *msg* is *msg.T*. ($T_i[j]$ and *msg.T*$[j]$, respectively, denote the jth component of these time vectors.)

The logical time at a process (site/node) evolves as follows:

(a) When an internal event or a send event occurs at process i, $T_i[i] := T_i[i] + 1$.

(b) Every message contains the current clock value, T_i, of its sender process i.

Correspondence to: M. Singhal, Department of Computer and Information Science, The Ohio State University, 2036 Neil Avenue Mall, Columbus, OH 43210, USA.

(c) When process i receives a message *msg*, then $\forall j$ do,

if $j = i$

then $T_i[j] := T_i[j] + 1$,

else $T_i[j] := \max(T_i[j], msg.T[j])$.

Thus the jth component of the time vector at a process reflects the highest value of the jth component of all timestamped messages it has received. Note that only $T_i[i]$ reflects the local activities at process i and $T_i[j]$ reflects what process i knows about the local timestamp (i.e., activities) of process j. Thus, time vector T_i reflects what process i knows of the latest state (local time) of all other processes. Note that each component of a time vector progresses independently.

An ordering relation "$<$" between vector timestamps is defined as follows: $T_i < T_j$ iff $(\forall k)$, $T_i[k] \leqslant T_j[k]$ and $T_i \neq T_j$. Note that $T_i < T_j$ *iff* the event at T_i happened before [7] the event at T_j. Thus, vector clocks [5,10] characterize the exact causal order among distributed events [14] as a partial order which is weaker than the total order of Lamport's clocks [7]. The system of vector time is a viable tool for analyzing the dynamics of distributed systems because it is capable of identifying causal ordering among events in a distributed system. The concept of vector clocks/timestamps has been indirectly used before in file consistency [13], distributed debugging [4], distributed mutual exclusion [15], distributed recovery [16], implementing causal distributed shared memory [1], execution tracing for testing distributed software [8] and in causal broadcast/multicast in the ISIS system [2].

However, the system of vector clocks is inefficient because the size of a timestamp is proportional to the number of processes in the system. (Charron-Bost has mathematically shown that causal order preserving clocks must have length N, the number of processes in the system [3].) If the number of processes is large, each message will be large and may have to be split into many packets. In this paper, we present an efficient method to maintain vector clocks which can cut down the communication overhead substantially when FIFO communication channels are available. The method reduces the size of timestamp related information to be transferred in a message at the expense of very little extra storage space at processes. The method is resilient to changes in the topology of the system such as the addition and deletion of communication edges.

2. The technique

A process i keeps two additional vectors, viz., $LU_i[1..N]$ and $LS_i[1..N]$ which are respectively called "Last Update" and "Last Sent". Entry $LU_i[j]$ indicates the value of $T_i[i]$ when process i last updated entry $T_i[j]$. Clearly, $LU_i[i] = T_i[i]$ at all times. $LU_i[j]$ needs to be updated only when the receipt of a message causes i to update entry $T_i[j]$. Entry $LS_i[j]$ indicates the value of $T_i[i]$ when the process i last sent a message to process j. It needs to be updated only when process i sends a message to process j. We make the following interesting observation:

Observation. Since the last communication from process i to process j, only those elements $T_i[k]$ have changed for which $LS_i[j] < LU_i[k]$.

Therefore, when process i sends a message to j, it only needs to send those entries $T_i[k]$ to process j for which $LS_i[j] < LU_i[k]$.

Our implementation of this observation needs just $O(n)$ space at each process. Another implementation of vector clocks [11] requires each process to keep the latest vector sent to every other process, thus, consuming $O(n^2)$ space at each process.

In the proposed technique, a process i needs to send to process j only a set of tuples $\{(x, T_i[x]) \mid LS_i[j] < LU_i[x]\}$ instead of a vector of N entries in a message. Note that this can substantially reduce the size of information sent in a message to update vector clocks because only a fraction of entries of T_i are likely to be modified between two successive message transfers to a process. This technique is specially effective when the number of processes is large because only a

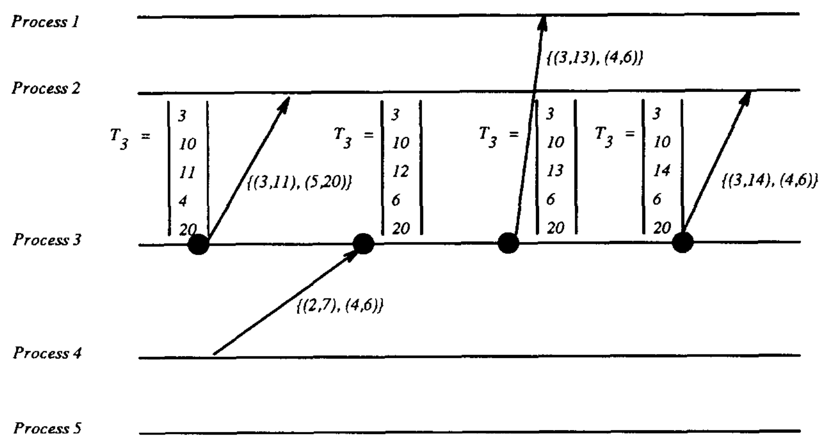

Fig. 1. Timing diagram for a five-process system.

few of the processes are likely to interact resulting in the update of only a few entries in the vector clock between two successive message transfers to a process [1].

[1] A further optimization, however small, is that process i never needs to send $(j, T_i[j])$ to process j.

We explain the technique with an example of a five-process system shown using Fig. 1 and Fig. 2. Figure 1 shows a timing diagram for the interactions of process 3 with the other processes. State 1 in Fig. 2 shows the data structure at process 3 at $T_3[3] = 10$. Process 3 needs to send a message to process 2. So it updates $T_3[3]$ to 11, and notes that since the last time it sent a message to

index	T_3	LU_3	LS_3
1	3	2	10
2	10	5	6
3	**10**	**10**	-
4	4	4	7
5	20	9	3

State 1

index	T_3	LU_3	LS_3
1	3	2	10
2	10	5	11
3	**11**	**11**	-
4	4	4	7
5	20	9	3

State 2

index	T_3	LU_3	LS_3
1	3	2	10
2	10	5	11
3	**12**	**12**	-
4	6	12	7
5	20	9	3

State 3

index	T_3	LU_3	LS_3
1	3	2	13
2	10	5	11
3	**13**	**13**	-
4	6	12	7
5	20	9	3

State 4

index	T_3	LU_3	LS_3
1	3	2	13
2	10	5	14
3	**14**	**14**	-
4	6	12	7
5	20	9	3

State 5

Fig. 2. An example of the data structure to maintain vector timestamps. The states of the data structure at process 3 from $T_3[3] = 10$ to $T_3[3] = 14$ in a five-process system.

process 2, only $T_3[3]$ and $T_3[5]$ have changed. This can be deduced from the current data structure because only $LU_3[3]$ which has now been incremented to 11 and $LU_3[5]$ which is 9 are greater than $LS_3[2]$ which is 6. Thus, the message to process 2 contains the timestamp {(3, 11), (5, 20)}. Process 3 now updates $LS_3[2]$ to 11, yielding State 2 shown in Fig. 2. Process 3 then receives a message with timestamp {(2, 7), (4, 6)} from process 4. By following the method for handling vector clocks (discussed in Section 1), process 3 updates entries $T_3[3]$ and $T_3[4]$ in T_3 to 12 and 6 respectively. Process 3 also updates entries $LU_3[3]$ and $LU_3[4]$ to 12, the current value of $T_3[3]$. This yields State 3 which is shown in Fig. 2. Process 3 now sends messages to processes 1 and 2 with timestamps as shown in Fig. 1 and the resulting data structures are shown as States 4 and 5, respectively, in Fig. 2.

3. Discussion

Impact on applications

Many distributed applications require observation of the latest global state of the system. Examples are implementing causal distributed shared memory [1], detection of inconsistent copies in a system with replicated data [13], distributed deadlock detection [6], and enforcement of mutual exclusion [15]. Algorithms for such applications can advantageously use the proposed technique because the "incremental" changes in the timestamp at a process are sufficient to construct the latest global state (assuming FIFO message delivery). Algorithms for termination detection [9] need to observe a consistent global state of the system and algorithms for recovery [16] need to construct a consistent global state of the system. Such algorithms can use the proposed technique for the same reason.

Several distributed applications require enforcement of "causal consistency" for point-to-point communications [14]. The ISIS distributed system environment [2] enforces causal consistency for multicast and other broadcast communications. Algorithms for such applications can use the proposed technique because the "incremental" changes in the timestamp at a process are sufficient to enforce causal consistency (assuming FIFO message delivery).

Some applications, such as prototyping language design, require that causal relationships be determined between two messages m_j and m_k arriving at a process from processes j and k, respectively [12]. Since only differential vectors are sent in the proposed technique, neither message timestamp *may* contain a tuple for the other sender even though the two send events may be causally related. To determine causality, the receiver process must be able to reconstruct the timestamps at processes j and k when m_j and m_k were, respectively, sent by them. This can be achieved in the proposed scheme using the following simple method which requires $O(n^2)$ space at each process: A process i maintains array $LR_i[1..N, 1..N]$ called "Last Received". Entry $LR_i[j, k]$ indicates the latest value $T_j[k]$ received by process i along a logical channel from process j. Vector $LR_i[j]$ indicates the last timestamp vector received from process j. When process i receives a set of tuples along a logical channel from process j, the corresponding entries in row $LR_i[j]$ are updated.

Locality of communications

In typical distributed systems, processes communicate in clusters over a very significant portion of the execution of a distributed application. In fact, application tasks are partitioned to minimize or localize communication. An example of the importance of locality is the vast literature on partitioning tasks onto sub-cubes of a hypercube. Decomposing a task hierarchically into a tree structure and mapping sub-tasks onto siblings in the tree also tends to localize communications among the children and sibling processes. Nested transactions exemplify this behavior, too. In all such cases, only incremental changes to the timestamp vector need to be transferred by processes, thus, making the proposed technique even more effective in such environments.

An optimization on storage requirements at a process can be performed. If the system topology is such that process i is not reachable from process j through a series of logical channels, then no event on process i can ever be causally affected by any event on process j [12]. Therefore, process i will never receive any information about process j's timestamp in any message and thus, $T_i[j]$ and $LU_i[j]$ will always be zero and undefined, respectively. Thus, process i need not keep an entry for all such processes j in its data structure T_i/LU_i, thus, saving on space. Also, if there is no logical channel from process i to process j, then $LS_i[j]$ will be undefined because process i never sends a message to process j. In such cases, process i need not keep an entry for process j in its data structure LS_i, thus, further saving on space.

4. Efficiency of the technique

We 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 timestamp related information to be transferred in a message as compared to when sending the entire vector timestamp. We also define the following terms:

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 clocks require $N.b$ bits of timestamp information, whereas the proposed technique requires $(\log_2 N + b).n$ bits. Thus, the efficiency of the technique is given by the following expression:

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

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

5. Conclusions

In this paper, we have presented an efficient technique to maintain Mattern/Fidge vector clocks, which cuts down the communication overhead due to propagation of vector timestamps by sending only incremental changes in the timestamp. The technique can cut down the communication overhead substantially if the interaction between processes is localized. The technique works in a system whose topology need not be known unlike the optimization presented in [12]. The technique has a small memory overhead (to store arrays LS_i and LU_i). 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

The authors have greatly benefitted from the comments of anonymous referees and F. Mattern and C. Fidge.

References

[1] M. Ahamad, P. Hutto and R. John, Implementing and programming causal distributed memory, in: *Proc. 11th Internat. Conf. on Distributed Computing Systems* (1991) 274–281.

[2] K. Birman, A. Schiper and P. Stephenson, Fast causal multicast, Computer Science Tech. Rept. TR-1105, Cornell University, April 1990.

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

[4] C.A. Fidge, Partial order for parallel debugging, in: *Proc. ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging* (1988) 183–194.

[5] C.A. Fidge, Timestamps in message-passing systems that preserve partial ordering, *Austral. Comput. Sci. Comm.* **10** (1988) 56–66.

[6] A.D. Kshemkalyani and M. Singhal, On characterization and correctness of distributed deadlock detection, OSU-CISRC-10/90-TR15 (under review for publication).

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

[8] S.K. Lloyd and P. Kearns, Using tracing to direct our

reasoning about distributed programs, in: *Proc. 11th Internat. Conf. on Distributed Computing Systems* (1991) 552–559.

[9] F. Mattern, Algorithms for distributed termination detection, *Distributed Comput.* **2** (1987) 161–175.

[10] F. Mattern, Virtual time and global states of distributed systems, in: M. Cosnard et al., eds., *Parallel and Distributed Algorithms* (North-Holland, Amsterdam, 1989) 215–226.

[11] F. Mattern, *Verteilte Basisalgorithmen* (Springer, Berlin, 1989), in German; ISBN 0-387-51835-5.

[12] S. Meldal, S. Sankar and J. Vera, Exploiting locality in maintaining potential causality, Tech. Rept. CSL-TR-91-466, Stanford University; also in: *Proc. ACM Symp. on Principles of Distributed Computing*, 1991.

[13] D.S. Parker, G.J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser and C. Kline, Detection of mutual inconsistency in distributed systems, *IEEE Trans. Software Engineering* **9** (1983) 240–247.

[14] A. Schiper, J. Eggli and A. Sandoz, A new algorithm to implement causal ordering, in: *Proc. 3rd Internat. Workshop on Distributed Algorithms*, Lecture Notes in Computer Science **392** (Springer, Berlin, 1989) 219–232.

[15] M. Singhal, A heuristically-aided algorithm for mutual exclusion in distributed systems, *IEEE Trans. Comput.* **38** (5) (1989).

[16] R. Strom and S. Yemini, Optimistic recovery in distributed systems, *ACM Trans. Comput. Systems* **3** (1985) 204–226.