A limitation of vector timestamps for reconstructing distributed computations

C. J. Fidge

Software Verification Research Centre, School of Information Technology, The University of Queensland, Queensland 4072, Australia.

Abstract

Vector timestamps provide a way of recording the causal relationships between events in a distributed computation. We draw attention to a limitation of such timestamps when used to reconstruct computations in which message overtaking occurred.

Key words: Distributed systems, vector time, logical clocks, timestamps, debugging

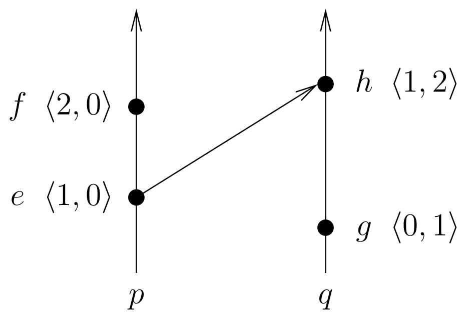
1 Causality in distributed systems

In a landmark article, Lamport [7] defined the causal relationships among events occurring in a message-passing distributed computation as the smallest relation '$\rightarrow$' such that

(1) if e and f are events in the same process, and e occurs before f, then $e \rightarrow f$,
(2) 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
(3) if $e \rightarrow f$ and $f \rightarrow g$, then $e \rightarrow g$.

2 Vector time

A number of researchers, most notably Mattern [8] and Fidge [4], later independently proposed *vector clocks* as a timestamping mechanism for distributed computations that captures causality. In a computation involving n parallel



f $\langle 2, 0\rangle$
e $\langle 1, 0\rangle$
h $\langle 1, 2\rangle$
g $\langle 0, 1\rangle$
p
q

Fig. 1. Simple example of vector timestamping.

processes, each process p maintains a logical clock vector of length 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 vector. For some vector $\vec{v}$ let $\vec{v}(i)$ denote its i^{th} 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.

(1) For each process p, all elements of $\vec{p}$ are initially 0.
(2) 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 $\vec{p}$.
(3) When process p performs a *send* event e, that produces a message m, it
  (a) 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 $\vec{p}$.
(4) 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 $\vec{p}$.

The vector timestamps associated with two *distinct* events e and f, from (not necessarily distinct) processes p and q, respectively, can then be used to determine if e and f are causally related, merely by comparing two elements, thanks to the following property [8].

$$e \rightarrow f \iff \vec{e}(p) \leq \vec{f}(p)$$

For example, Figure 1 shows the timestamps associated with four events in a simple computation involving two processes which exchange one message. These timestamps tell us that $g \rightarrow h$ because $1 \leq 2$, and $e \rightarrow h$ because $1 \leq 1$.

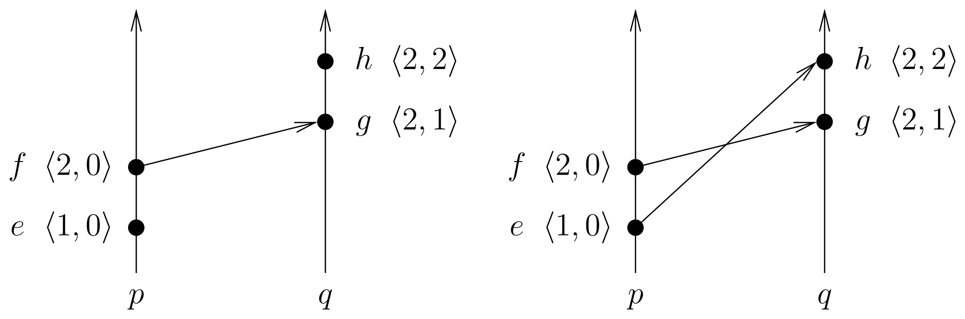


Fig. 2. Two distinct computations with identical timestamps.

We can also determine when events are *not* causally related, for instance, $f \not\to e$ because $2 \not\leq 1$, $f \not\to g$ because $2 \not\leq 0$, and $g \not\to f$ because $1 \not\leq 0$. (Events f and g are thus 'concurrent' or 'independent.')

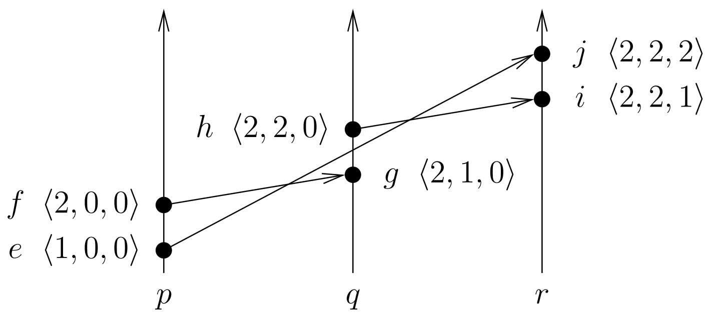
3 Applications

Vector clocks offer significant advantages over other timestamping mechanisms, most notably independence from absolute timing, and the ability to recognise the *absence* of causality [6]. They have therefore been used in a number of applications including detection of global states, enforcement of causal ordering, and concurrent software metrics [5].

For debugging distributed programs, vector timestamps offer a way of reconstructing a computation after it has occurred. The timestamps can be logged at run time, and the computation then reconstructed later for leisurely post-mortem analysis. For instance, the four timestamps shown in Figure 1, namely $\langle 1,0\rangle$, $\langle 2,0\rangle$, $\langle 0,1\rangle$ and $\langle 1,2\rangle$, are sufficient for a simple tool to reconstruct and display this computation. Indeed, Figure 1 is the *only* computation that can be drawn consistent with these timestamps. (We assume the vertical displacement between events is irrelevant [7].) In particular, note that timestamp $\langle 1,2\rangle$, associated with event h, tells us that a message was sent to process q by process p, because the first vector element indicates knowledge of the occurrence of one event in process p.

4 A limitation

It is tempting, therefore, to conclude that a set of vector timestamps, one per event, fully characterise a distributed computation. However, we observe that in systems that allow message 'overtaking' this is not necessarily so. Figure 2 shows two distinct computations that have identical event timestamps. On



j $\langle 2,2,2\rangle$
i $\langle 2,2,1\rangle$
h $\langle 2,2,0\rangle$
g $\langle 2,1,0\rangle$
f $\langle 2,0,0\rangle$
e $\langle 1,0,0\rangle$
p
q
r

Fig. 3. A computation with indirect message overtaking.

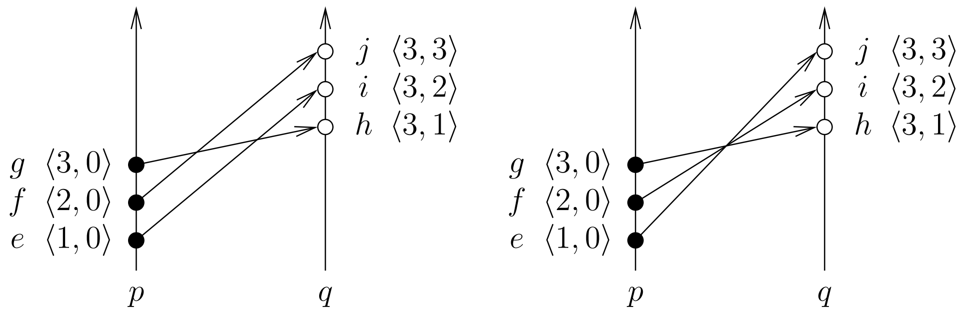


Fig. 4. Two computations with send and receive events marked differently.

the left events e and h are internal to processes p and q. However, in the computation on the right event e is a send, and event h 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.

This phenomenon occurs whenever overtaking of information transference is possible, even indirectly. Figure 3 shows a computation in which indirect communication from process p to r, from event f to event i, via events g and h, overtakes the direct message from event e to j. In this case the timestamps assigned would be the same if e and j were internal events. Thus, merely prohibiting overtaking of messages between *pairs* 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. Certainly this information would be sufficient to disambiguate the computations in Figure 2. However the two computations in Figure 4, in which send and receive events have been distinguished, show that this is insufficient in general. Corresponding events in both computations receive the same timestamps, and have the same 'types,' but the computations are different.

5 Cause

The cause of this ‘problem’ is straightforward. Step 4b, Section 2, is responsible for merging causality information obtained through the receipt of a piggybacked message timestamp with the local process clock. However, when the received message has previously been overtaken this step has no effect. The update to the process clock due to a receive event (Step 4) is then identical to that performed for an internal one (Step 2), hence the ambiguity. (Interestingly, the more expensive *matrix clock* model [11] also suffers from the same problem.)

This is not a weakness or error in the vector time algorithm, however. It is, in fact, a natural consequence of the definition of causality for distributed systems. Property 3, Section 1, tells us that causality is transitive. Consequently, overtaken messages do not contribute new causal relationships. For instance, the definition of causality states that the computation on the left in Figure 2 defines the following causal relationships: $e \to f$, $e \to g$, $e \to h$, $f \to g$, $f \to h$ and $g \to h$. Exactly the same set of relationships is created by the computation on the right, so it is to be expected that both computations are timestamped identically.

Previously, Charron-Bost et al. noted this same phenomenon when discussing causality from a graphical perspective [2]. They observed that a space-time diagram contains more information than a Hasse diagram formed from the causal relationships between events. The two computations illustrated by the space-time diagrams in Figure 2 both have the same Hasse diagrams and are thus indistinguishable when only event causality is considered.

6 A solution

Fortunately there is a straightforward solution. We noted in Step 3c, Section 2, that each message carries with it the timestamp corresponding to the sending event. Since vector timestamps are unique throughout a computation, this information is sufficient to unambiguously determine the pattern of communications. Thus, if receive events are logged as *pairs* of timestamps, consisting of the piggybacked time and the ‘local’ time, then the two computations in Figure 4 would be distinguishable. In the left-hand computation the receive events would be logged as h: $(\langle 3,0\rangle, \langle 3,1\rangle)$, i: $(\langle 1,0\rangle, \langle 3,2\rangle)$, and j: $(\langle 2,0\rangle, \langle 3,3\rangle)$. In the right-hand computation the events would be logged as h: $(\langle 3,0\rangle, \langle 3,1\rangle)$, i: $(\langle 2,0\rangle, \langle 3,2\rangle)$ and j: $(\langle 1,0\rangle, \langle 3,3\rangle)$. Matching receive events with their corresponding sends is now trivial and a debugging tool can easily display the two distinct computations. (When comparing events to detect causal relationships, using the property described in Section 2, the first element of the pair is ignored.) Of course, the disadvantage is that the information that must be logged when a receive event occurs has been doubled.

Indeed, this solution is not surprising. The principle of making use of the piggybacked timestamps to identify message overtaking is well established in the vector time community. It is the basis of algorithms for enforcing *causal ordering*, in which message overtaking is prevented by checking piggybacked timestamps and accepting received messages only when it is clear that there are no causally-preceding messages that have not yet arrived [10].

Other ways of solving the problem are possible, of course. The key to the solution above 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, will be equally effective.

7 Related work

In this paper we are concerned with being able to statically recreate the relationships between events in a past computation. A related, but distinct, problem is that of dynamically recreating the computation itself. Debugging mechanisms for distributed or concurrent programs achieve such a *replay* capability by recording traces containing sufficient information to ensure that, when reexecuted with the same inputs, the program can be forced to follow the same control paths again. Typically these traces record the nondeterministic choices made within a process [3], the order in which messages were accepted by a process [9], or both [1]. However, such traces do not contain as much information as vector timestamps, so neither the stored traces, nor the replayed program, directly model event causality.

8 Conclusion

We have described a limitation of vector clock timestamps for characterising distributed computations, and have examined its cause and solution. While not a profound issue, developers of debugging and analysis tools for distributed systems should nevertheless appreciate this property of causality.

Acknowledgements

I wish to thank the anonymous referees for pointing out a number of relevant references, the students of the University of Queensland's Distributed Computing course for helpful discussions, and Andrew Martin for reviewing a draft of this paper. This work was funded, in part, by the Information Technology Division of the Australian Defence Science and Technology Organisation.

References

[1] H. S. Bae, H. S. Kim, and Y. R. Kwon. An efficient debugging method for message-based parallel programs using static analysis information. In *Proceedings Asia-Pacific Software Engineering Conference (APSEC'95)*, pages 96–105, Brisbane, December 1995.

[2] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9(4):173–191, 1996.

[3] C. J. Fidge. Reproducible tests in CSP. *The Australian Computer Journal*, 19(2):92–98, May 1987.

[4] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In K. Raymond, editor, *Proceedings of the 11th Australian Computer Science Conference (ACSC'88)*, pages 56–66, February 1988.

[5] C. J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, August 1991.

[6] C. J. Fidge. Fundamentals of distributed system observation. *IEEE Software*, 13(6):77–83, November 1996.

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

[8] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard et al., editors, *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[9] R. H. B. Netzer and J. Xu. Adaptive message logging for incremental program replay. *IEEE Parallel & Distributed Technology*, 1(4):32–39, November 1993.

[10] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, September 1991.

[11] M. Raynal and M. Singhal. Capturing causality in distributed systems. *IEEE Computer*, 29(2):49–56, February 1996.