SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 97-43

A limitation of vector timestamps for reconstructing distributed computations

C. J. Fidge

December 1997

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

Note: Most SVRC technical reports are available via anonymous ftp, from `svrc.it.uq.edu.au` in the directory `/pub/techreports`. Abstracts and compressed postscript files are available via `http://svrc.it.uq.edu.au`

A limitation of vector timestamps for reconstructing distributed computations

C. J. Fidge

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.

1 Causality in distributed systems

In a landmark article, Lamport [5] 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 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 [6] and Fidge [2], 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. 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

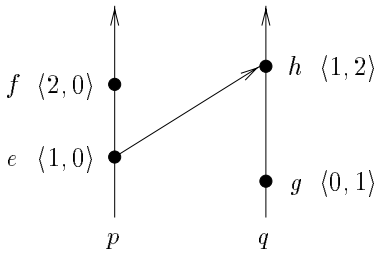

Figure 1: Simple example of vector timestamping.

(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 [6].

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

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

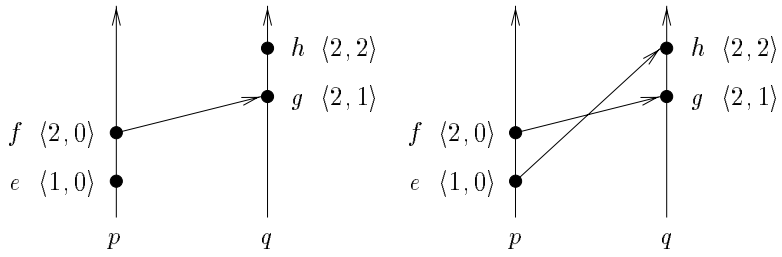

Figure 2: Two distinct computations with identical timestamps.

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 [4]. They have therefore been used in a number of applications including detection of global states, enforcement of causal ordering, and concurrent software metrics [3].

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, $\langle 1,0\rangle$, $\langle 2,0\rangle$, $\langle 0,1\rangle$ 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 [5].) 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 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

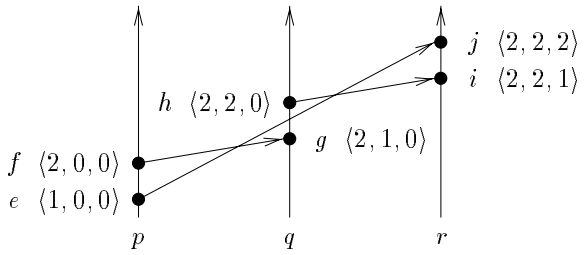

Figure 3: A computation with indirect message overtaking.

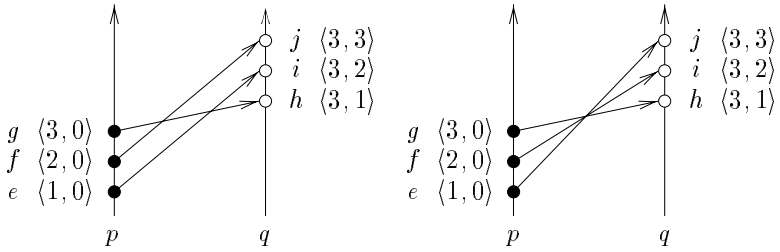

Figure 4: Two computations with send and receive events marked differently.

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.

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: $e \rightarrow f$, $e \rightarrow g$, $e \rightarrow h$, $f \rightarrow g$, $f \rightarrow h$ and $g \rightarrow h$. 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.

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 messages received only when it is clear that there are no causally-preceding messages that have not yet arrived [1, 7].

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

[2] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):56–66, February 1988.

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

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

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

[6] 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.

[7] M. Raynal and A. Schiper. The causal ordering abstraction and a simple way to implement it. Technical Report 1132, Institut National de Recherche en Informatique et en Automatique, France, December 1989.