Fundamentals of Distributed System Observation

Colin Fidge
Software Verification Research Centre,
Department of Computer Science,
The University of Queensland,
Queensland 4072, Australia.
Telephone: +61 7 3365 1648
Facsimile: +61 7 3365 1533
E-mail: cjf@cs.uq.edu.au

Abstract

Merely observing activity in distributed systems is challenging: even an observer's location influences its interpretation of messages notifying it of significant events in the network. Various event timestamping mechanisms have been proposed to help overcome this effect. This tutorial firstly identifies those problems that may arise when relying on the arrival order of notifications. It then critically examines four timestamping models in terms of how well they solve the observability problems, thus providing insight into their practical abilities.
KEYWORDS AND PHRASES: distributed systems, timestamps, logical time, vector time, causality

Introduction

Testing and debugging a distributed system presents the programmer with profound challenges: merely observing what is happening in a network of processes is difficult. In this tutorial I review the issues and then use this characterisation to critically analyse the power of timestamp-based solutions, thus providing insight into the expressibility of different time models.

Definitions

The effects discussed in this article usually manifest themselves in *distributed* systems, that is, those in which there are a number of communicating, spatially-separated processors. However, the concepts are applicable to any system exhibiting *concurrency*, or the appearance of two or more events occurring simultaneously, including multiprocessor machines and uniprocessor multi-tasking.

No assumptions are made about the nature of *events* occurring in the system other than to note that they represent discrete actions meaningful to the programmer. They may be the execution of single machine instructions or entire procedures; the level of granularity is unimportant.

For brevity in this presentation assume that *asynchronous message-passing* is the only medium for interprocess communication in the system under test—senders do not block and messages are buffered until a receiver requests one (but FIFO queuing is not necessarily assumed). However, the concepts extend straightforwardly to other forms of communication, such as synchronous message-passing, shared memory, remote procedure calls or rendezvous.

The motivating concern is the *observability problem* in distributed systems. This is the difficulty of attempting to determine the order in which events occurred during a given *computation* (i.e., a single 'execution' or 'run' of the system, definable by the particular set of control paths followed on this occasion). Let us adopt *causality*, the ability of one event to affect another, as the basis for defining event *order* because it allows us to reason independently of any particular time-frame [10].

An *observer* of a distributed system is any entity that attempts to examine a computation. Observers may be human programmers, watching an animated display of system activity, or other processes in the network, automatically analysing system activity. As shown in Figure 1, observers may watch the system 'live', while the computation is in progress, or examine a post-mortem event log or trace. In each scenario the observer must be informed whenever interesting events occur. In practice this involves instrumenting the system under observation with *probes* that send *notification* messages to the observer (or write entries into the log) following the occurrence of such events.

The concern in this article is to review how accurate a view of system activity is presented to the observer by these notifications. I assess a number of proposed methods of event *timestamping*. These provide a way of associating a number with each event that can be included in the notification messages and used by the observer to assess event orderings.

As a simple running example, let us use the computation shown in Figure 2. It shows a system consisting of two parallel processes P and Q. Process P performs two events. First, event a denotes the transmission of a message. P then performs an action local to itself, denoted b. Process Q also performs two events, denoted c and d, the first of which is the reception of the message sent by P.

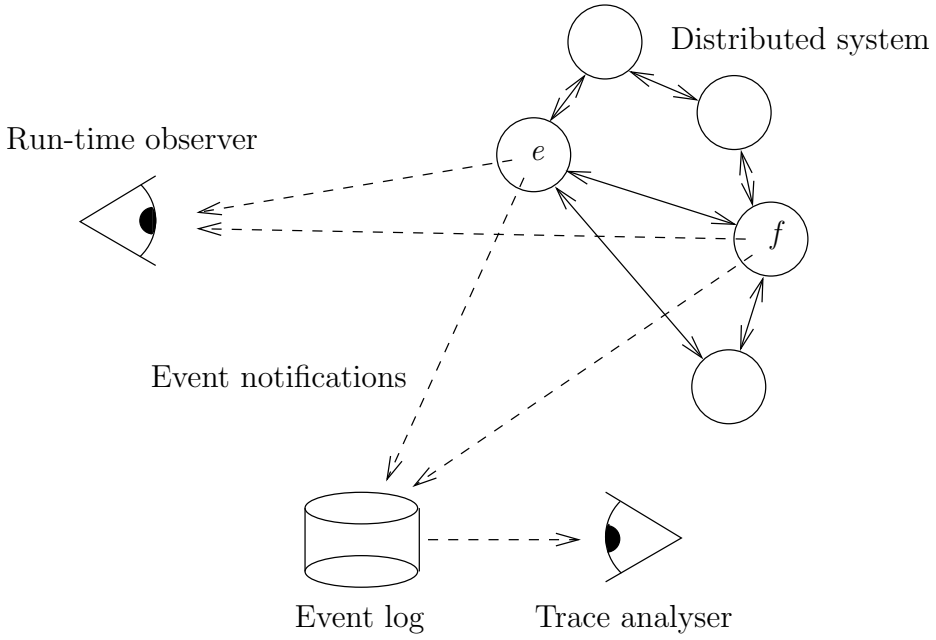

Figure 1: Observers of events e and f in a distributed computation.

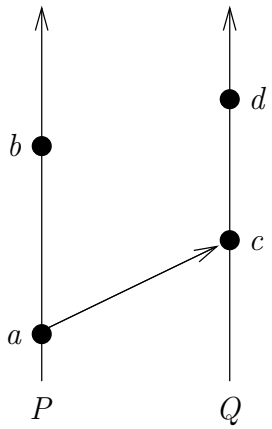

Figure 2: A distributed computation.

It is important to note that, in the absence of any further information, this same computation can be redrawn as shown in Figure 3. Here the same events occur in the same relative *local* orders. Only the omnipotent viewpoint provided by such diagrams allows us to know that independent events b and c are interleaved differently; processes P and Q cannot 'see' any difference. Lamport [10] discusses this equivalence in depth.

Fundamental observability effects

When an observer relies on the arrival order of notification messages to determine event orderings in a distributed system, four types of discrepancies can arise.

Multiple observers see different orderings

Whenever there are two or more observers of a particular computation they may each perceive different event orderings. In Figure 4 two observers O and R are notified of the occurrence of events b and c (notification messages are shown as dashed arrows). Due to the transmission delays associated with the messages, observer O believes that event b occurred before event c, whereas observer R sees event c occur before event b. Both interpretations are valid, but they cannot be easily reconciled. (There is an obvious parallel with spacetime physics—the observer's location determines its view of the universe.)

Incorrect perceived orderings

More seriously, the perceived event ordering may simply be incorrect. In Figure 5 observer R erroneously believes that event c occurred before event a. Such an effect may be caused by notifications being delayed due to retries or being routed to the observer through indirect pathways.

Same computation exhibits different orderings

When testing or debugging a system the programmer typically wants to replay the same computation several times in order to study different aspects of its behaviour. Unfortunately an observer may see different event orderings at each replay!

Figure 6 shows two different instances of the same computation: in both cases processes P and Q perform exactly the same events in the same relative orders (a then b and c then d). In the first instance observer O sees event c occur before event b, but in the second instance (involving the same program, supplied with the same data, and following the same control paths) the observer sees b occur before event c. This nondeterministic behaviour during debugging may be due to minor differences in the processor and link loads caused by other

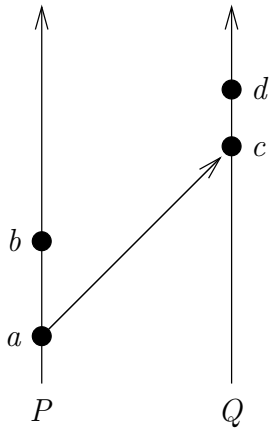

Figure 3: A different view of the same computation.

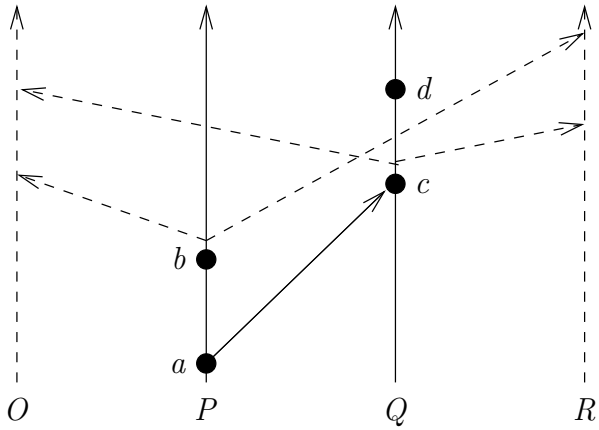

Figure 4: Multiple observers see different orderings.

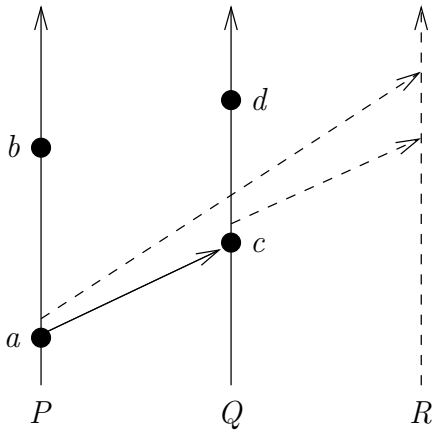

Figure 5: Incorrect perceived ordering.

system activity and may occur even when the system being observed is performing a deterministic computation! This can be a major source of frustration when debugging distributed systems because previously observed ordering errors may vanish, even though a replay mechanism is being used.

Arbitrary orderings adopted

Relying on the arrival time of notifications to determine event orderings also means that arbitrary orderings are assumed between unrelated events. In Figure 7 observer O first sees that event a occurred before event c. This is a valid observation in the sense that a *must* occur before c in this computation. Observer O then sees event c occur before event b. This perceived ordering is merely an artefact of the notification mechanism. As shown by comparing Figures 2 and 3, events b and c are independent in this computation and may occur in either order (in a global sense of time). This is a serious problem because such arbitrary orderings are *indistinguishable* from genuine 'enforced' orderings and thus inhibit the observer's ability to know if the same event orderings will be maintained in future tests. During debugging, a programmer observing c preceding b may mistakenly conclude that this program has sufficient interaction between processes P and Q to always maintain this relationship.

Effectiveness of timestamping

To accurately observe behaviour in a distributed system more information is needed than just the arrival order of notifications. An obvious approach is to *timestamp* the events of interest and send this information to the observer in the notification messages. The observer can then use these values to determine the true order in which events occurred.

The remainder of this section critically analyses the ability of four different timestamping mechanisms to resolve the observability effects described above. Table 1 summarises the results; a '✓' indicates that the proposed timestamping mechanism satisfactorily overcomes the effect process distribution may have on observability.

Local real-time clocks

An obvious approach is to make use of whatever real-time clock is available in the hardware of each processor as the source of timestamps. All notification messages then have the same time value associated with each distinct event. This means that all observers see the same time orderings, thus avoiding the first effect. Unfortunately the others persist. Figure 8 shows two possible ways in which the events in our example computation may be timestamped. Since the clocks on different processors are not synchronised they will inevitably drift, so it is

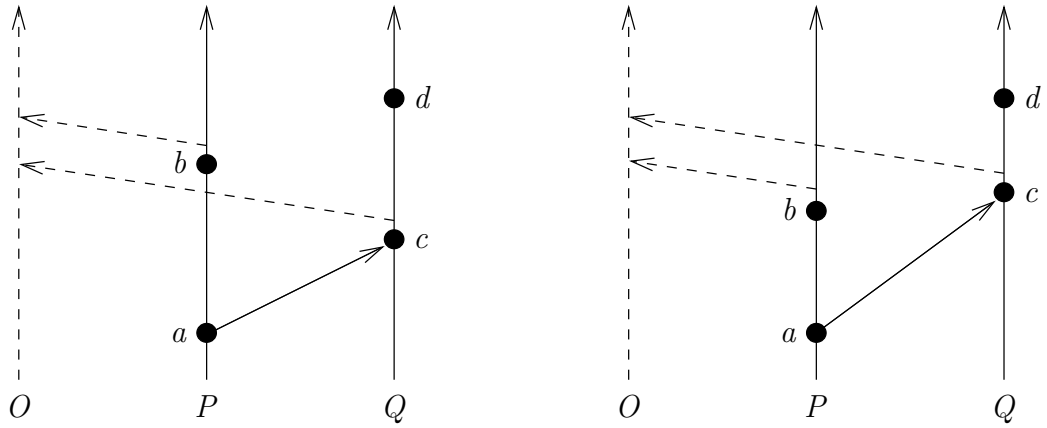

Figure 6: Same computation exhibits different orderings.

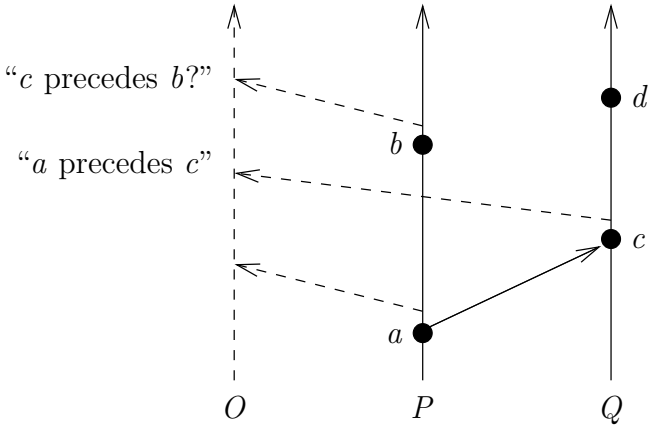

Figure 7: Arbitrary orderings adopted.

	Ordering mechanisms				
		Real-time timestamps		Logical timestamps	
Effects	arrival ordering	local clocks	global clock	totally ordered	partially ordered
Multiple observers see different orderings		✓	✓	✓	✓
Incorrect perceived orderings			✓	✓	✓
Same computation exhibits different orderings				✓	✓
Arbitrary orderings adopted					✓

Table 1: Effectiveness of timestamping mechanisms.

sidebar:

A closed-world assumption

For the logical timestamping mechanisms discussed in this article to accurately model event relationships two conditions are essential.

- All processes in the system under observation must participate in the timestamping algorithm.
- All forms of interaction between processes must be timestamped [4].

If any process does not propagate timestamps correctly, or if the processes can interact through some covert channel, e.g., the file system, that is not timestamped, then the clock values may not accurately reflect causal relationships [10].

Conversely, a passive observer process must *not* participate in the timestamping algorithm if it is to be unintrusive. If the observer propagates timestamps it receives in notification messages, then the mere act of notifying the observer creates detectable causal relationships that would not exist in the observer's absence.

not meaningful to compare times across machine boundaries. Incorrect orderings may therefore be seen; on the left of Figure 8 the clock on P's processor is ahead of that of Q so event c erroneously appears to occur before event a. Also, each instance of the same computation may receive different timestamps, as shown by the two scenarios in Figure 8. Finally, the ordering between independent events, such as b and d in Figure 8, is randomly influenced by processor loads and the (in)ability of the clocks to remain synchronised.

A global real-time clock

As an improvement let us assume that the clocks are synchronised throughout the distributed system to a high degree of accuracy, in effect providing a global reference for real time. This avoids the effect of incorrect orderings being perceived; time readings become meaningful across processor boundaries and hence always reflect the actual order of event occurrence. Nevertheless, as shown by Figure 9, the same computation may still yield different orderings if system loads vary between tests, and arbitrary orderings are still imposed on independent events.

It is perhaps surprising that such a powerful facility as global real time, which is expensive to achieve, still fails to satisfy our needs. To answer the question of whether one event *must* precede another in a particular computation with certainty an unbounded number of tests would be required!

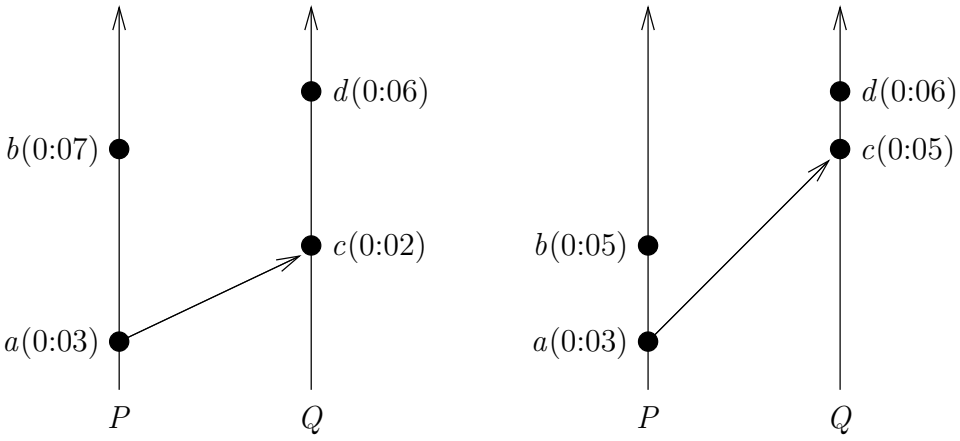

Figure 8: Real-time timestamps using (unsynchronised) local clocks.

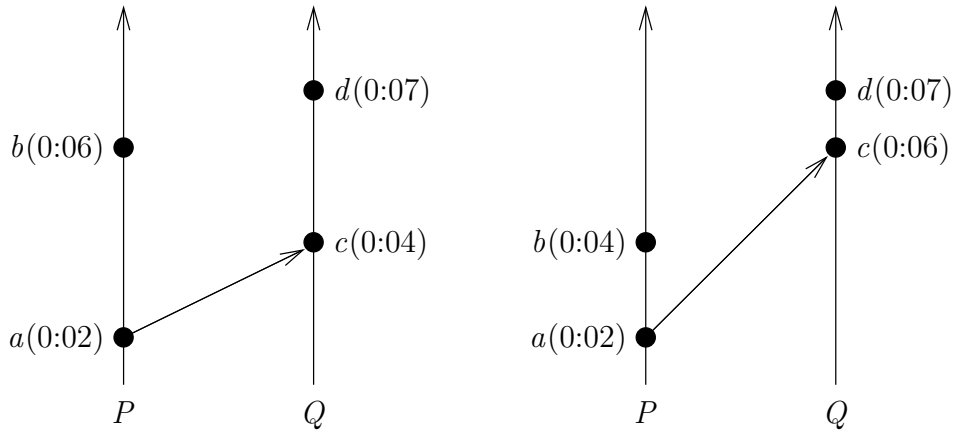

Figure 9: Real-time timestamps using a global clock.

Totally ordered logical clocks

The issues remaining above are due to the use of *absolute* time to order events. These values are randomly influenced by factors such as processor loads and the time at which each process is started. As a solution to this, *logical* clocks have been proposed as a more objective ordering mechanism. A simple system of logical clocks can be used to totally order the events in a distributed system using the following rules [10]:

- each process maintains an integer counter,
- whenever a process performs an event of interest it increases its counter value,
- whenever a process sends a message the current counter value is 'piggybacked' on the message, and
- whenever a process receives a message it sets its own counter to be greater than its current value and that of the piggybacked value received.

Figure 10 shows the totally ordered timestamps associated with each event in our example computation, assuming that the counters start from zero and are incremented by one at each event occurrence. The values for events a and b are obvious. The receive event c, however, is given timestamp 2, rather than 1, because it must have a higher value than the corresponding send event.

The timestamps thus generated are not unique, as shown by events b and c. The total ordering is completed by adopting an arbitrary, but consistent, ordering among the processes when two events have the same timestamp [10].

This mechanism has the same advantages as global real-time clocks and also precludes the possibility of the same computation producing different orderings. The timestamps are consistently associated with each event no matter how many times the computation is replayed, regardless of differences in absolute timing (assuming the algorithm for increasing timestamps is deterministic). This is an important advantage during testing and debugging because it avoids the need to re-perform the same computation in order to see if different orderings are observable. (A nondeterministic program may still generate several distinct computations from the same input data, however.) For this reason, and the ease of implementing them, totally ordered logical clocks have been used in many distributed debugging systems.

One issue remains however. An arbitrary ordering is still imposed on independent events. An observer relying on the timestamps in Figure 10 will mistakenly conclude that b always occurs before d, even though there is no interaction between processes P and Q to guarantee this. This misleading view will thwart any attempts to identify problems stemming from inadequate synchronisation between events.

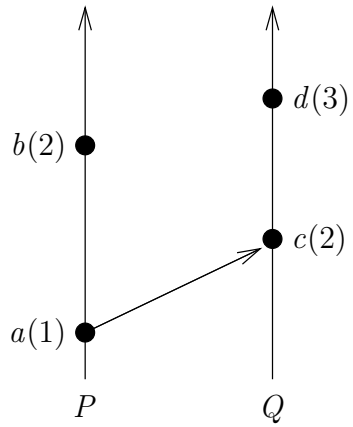

Figure 10: Logical timestamps using a totally ordered clock.

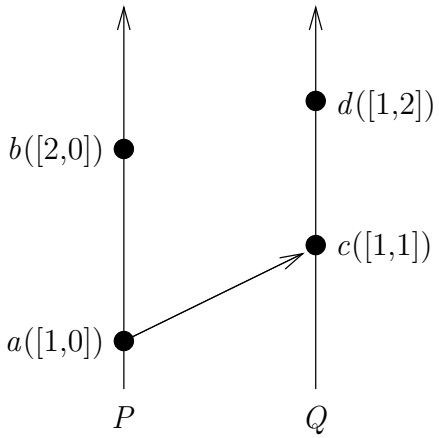

Figure 11: Logical timestamps using a partially ordered clock.

Partially ordered logical clocks

The ordering of events defined by totally ordered clocks is an incomplete view of event causality. However a straightforward extension allows all causal orderings to be preserved. *Partially* ordered logical clocks can be maintained as follows [6, 11]:

- each process maintains an array of counters, with one element in the array for every process in the distributed system,
- whenever a process performs an event of interest it increases its own counter value in its array,
- whenever a process sends a message the array of counters is piggybacked on the message, and
- whenever a process receives a message it sets each element in its own array to be the maximum of the current value of the element and the corresponding element in the piggybacked array received.

Figure 11 shows how our example would be timestamped. Processes P and Q both maintain an array of two counters. In each array the first counter value represents the number of events known to have occurred in process P and the second value represents the number of events known to have occurred in process Q. (This example has a fixed number of processes, but the concept extends to dynamic process creation [6].)

The entire array forms the timestamp. When comparing two such timestamps we can conclude that some event e, occurring in process i, preceded some event f, occurring in process j, only if

1. event f's timestamp has a counter value for process i greater than or equal to the counter for process i in event e's timestamp, and
2. event e's timestamp has a counter value for process j strictly less than that for process j in event f's timestamp.

The second condition avoids reflexivity [10] and allows for the case of synchronous message-passing where corresponding send and receive events are treated like a single joint action and receive the same timestamp [6]. A simpler test is available if we know that only asynchronous message passing will be used [11].

For instance, in Figure 11 we can conclude that event a preceded event d because d knows of the occurrence of 1 event in process P, as does a, but a does not know of as many events (0) in process Q as d (2). Similarly we know that c preceded d because d knows of more events in process Q (2) than does c (1).

These observations can also be achieved using totally ordered clocks. However, where the totally ordered model assumed that b preceded d, the partially ordered model does not. We cannot show that b precedes d because b knows of more

events (2) in process P than does d (1). Furthermore, we cannot show that d precedes b either because d knows of more events occurring in process Q (2) than does b (0). An observer can therefore make use of these timestamps to determine that events b and d are *unordered*; they are independent actions that (in global time) may occur in either order, or even simultaneously.

This capability overcomes the last of our outstanding observability effects (see Table 1). Partially ordered clocks reflect only 'true' causal orderings and make the *absence* of ordering explicit.

Discussion

Synchronous notifications

Many of the observability effects defined above stemmed from unpredictable delays between the time events occurred in the distributed system and the time the observer received a notification. It is therefore tempting to assume that using *synchronous* communication between the system and its observer(s) will avoid these effects. Unfortunately, as shown in Figure 12, the problems persist. (Information is still being sent from processes P and Q to O, but the double-headed arrows denote the bidirectional causality relation that results from synchronous communication.)

It is still possible for the arrival time of notifications to incorrectly reflect event orderings. In Figure 12 process P is delayed after performing event a, perhaps due to contention for the processor, before it can send the event notification to O. Consequently, the notification for event c arrives before that of its causal predecessor a. Also, arbitrary orderings are still imposed, as is the case between b and d. Similarly, multiple observers may see different orderings and the same computation may yield different observations.

Intrusive observers

Thus synchronous notification messages cannot solve our observability problems. Even worse, synchronous notification introduces a form of *probe effect* (see sidebar) in which the mere act of observing the system alters its behaviour! (Again there is a parallel with quantum physics.)

This manifests itself in two ways. Firstly, processes which wish to notify the observer are effectively blocked until the observer deigns to communicate with them. This may alter real-time behaviour and nondeterministic choices in the system under observation. Also, any bias on the observer's part about which system processes it 'prefers' to communicate with will influence their ability to proceed.

Secondly, the bi-directional causality relationship defined by synchronous communication creates new causal orderings that would not exist in the absence of

sidebar:

The probe effect

An issue often associated with, but distinct from, the observability problem is the *probe effect* (sometimes referred to as the 'Heisenberg effect' by aspiring physicists). This is the danger that auxiliary code added by a debugger will alter the behaviour of a concurrent program under study [5]. Whereas the observability problem concerns the ability to study a particular computation, the probe effect concerns the ability to perform a given computation in the first place. The probe effect may make existing errors vanish, by preventing certain erroneous computations from occurring, or may cause new errors to appear, by allowing computations not possible in the original program. Many systems take extreme measures in an attempt to avoid the probe effect, typically by trying to account for the time occupied by the auxiliary code [1, 15]. Unfortunately it has long been recognised that software-based debugging utilities inevitably introduce some degree of intrusiveness [14]. (Customised hardware can be used to unintrusively monitor a system [12] but this is expensive and inflexible.)

The probe effect manifests itself by

- changing the probability of making particular nondeterministic choices,
- altering real-time execution speeds,
- changing access patterns to inadequately protected shared memory, or
- making a program augmented with debugging probes distinguishable from the unaugmented program.

A commonly-suggested solution is to permanently install debugging probes so that the program undergoing debugging is the *same* as the final 'production' version [12, 5, 8, 7], albeit with a penalty in terms of run-time overheads. (This approach has the benefit of leaving debugging 'hooks' in an operational system for tracking down infrequent errors that eluded the testing and debugging phases, but such access points may also be a security hazard!)

the observer. In Figure 13 each event is followed by a notification message. After receiving notification of event b in process P, the observer interacts with process Q, to receive notification of event c. This creates a causal link between P and Q, via the observer O, that means that event b potentially causally affects event d!

sidebar:

Reproducibility

It is important to clearly distinguish the probe effect from the difficulty of achieving *reproducibility* while observing concurrent software. Having seen the system perform some behaviour of interest, programmers need reproducibility to force this particular computation to occur again for closer inspection. However the problem of achieving reproducibility exists for any program that makes nondeterministic choices, regardless of the presence or absence of debugging probes, and can be treated using methods quite distinct from those proposed to overcome the probe effect [12]. These include recording traces for later replay [5] or giving the programmer explicit control over nondeterministic alternatives [9]. (Interestingly, the third 'effect' described in this article can be overcome by including the observer itself in a trace-based reproducibility mechanism.) Practical debugging problems attributed to the probe effect are, quite often, actually manifestations of the difficulty of achieving reproducibility.

sidebar:

Using logical clocks to control computations

This tutorial focusses on the use of timestamping to passively *observe* activity in a distributed computation. However timestamps are also used to *control* activity. In particular, for systems maintaining replicated data, partially ordered logical clocks have been used to implement

- *causally-ordered communication*, in which update messages are delivered in an order consistent with the causal order in which they were sent, and
- *totally-ordered communication*, in which multicast updates are guaranteed to be delivered to all recipients in the same order.

Many experimental systems following these principles exist, the most well-known being the ISIS distributed system toolkit [2].

Interestingly, the same observability issues are relevant in this context. These communication models are implemented by delaying receipt of a message until those messages that must precede it have been received, as determined by the timestamps. Thus *every* receiving process acts as an 'observer' in this case.

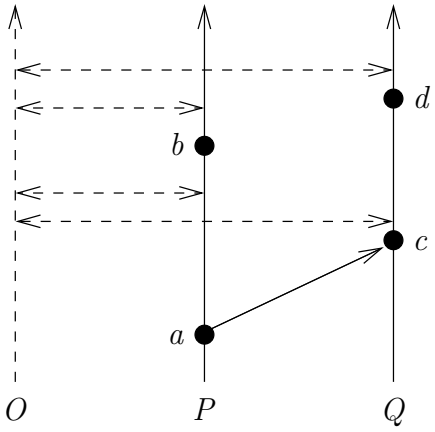

Figure 12: Inaccurate reporting using synchronous notifications.

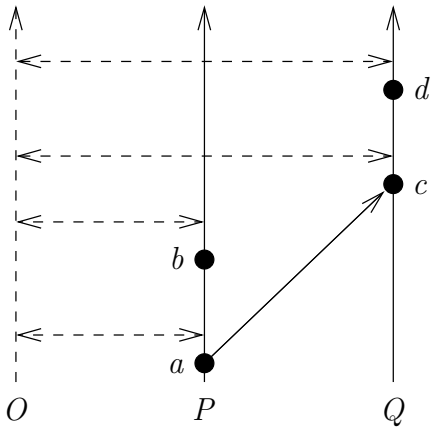

Figure 13: Intrusiveness due to synchronous notifications.

This is easily demonstrable by following arrows from b to d in Figure 13.

Conclusion

We have reviewed a number of mechanisms intended to give observers a view of the order in which events occur in a distributed computation. It was shown that partially ordered logical clocks are the only one capable of fully indicating ordering between events; all other timestamping mechanisms may misleadingly impose orderings between independent events.

This is not to say that partially ordered logical clocks are the only mechanism that should be used. They are notoriously expensive to implement [4]; the size of the array must be as great as the number of parallel processes [3]. Many schemes have been suggested for reducing their cost but all involve either some loss of causality information [6] or merely trade off storage requirements against communication and processing overheads [13]. Nevertheless, programmers using other timestamping mechanisms need to appreciate the limitations of the method used and understand that they are receiving an incomplete view of event ordering. Partially ordered clocks can then be used to give the complete picture when necessary.

sidebar:

Partially ordered logical clocks: A personal perspective

This research began in the mid-80's with the author's own attempt to debug a parallel program. The program drew a complex diagram on a graphics terminal. This was done using several processes, each in charge of their own portion of the screen, presided over by a controller process which initialised and closed the display surface. The program was found to occasionally crash towards the end of the run, seemingly at random. Poor synchronisation between the controller and its subordinates in the final stages of the execution was suspected but none of the debugging tools then available was capable of giving a view of system behaviour adequate to confirm this suspicion. (Ultimately it transpired that the problem was caused by the controller process performing its closing actions after receiving a 'finished' message from just *one* of its subordinates, instead of waiting for them all.) The frustrations encountered in this exercise led me to develop partially ordered logical clocks [6] as a model that can detect the *absence* of ordering. Indeed, there seemed to be a strong need for such a mechanism; several other researchers independently developed the same model to solve their own observability problems [13].

Acknowledgements

I wish to thank Mark Utting and the anonymous *ACSC* and *Software* referees for their insightful comments and corrections.

References

[1] F. Baiardi, N. de Francesco, and G. Vaglini. Development of a debugger for a concurrent language. *IEEE Transactions on Software Engineering*, SE-12(4):547–553, 1986.

[2] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53,103, December 1993.

[3] B. Charron-Bost. Concerning the size of clocks. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 176–184. Springer-Verlag, 1990.

[4] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communications. *Operating Systems Review*, 27(5):44–57, December 1993.

[5] P. S. Dodd and C. V. Ravishankar. Monitoring and debugging distributed real-time programs. *Software—Practice & Experience*, 22(10):863–877, October 1992.

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

[7] A. Gordon. *Ordering Errors in Distributed Programs*. PhD thesis, University of Wisconsin-Madison, 1985.

[8] D. Haban. DTM: A method for testing distributed systems. In *Proc. Sixth Symposium on Reliability in Distributed Software and Database Systems*, pages 66–73, Virginia, March 1987. IEEE Comp. Soc. Press.

[9] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems*, 5(2):121–150, 1987.

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

[11] 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.

[12] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

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

[14] H. Tokuda, M. Kotera, and C. Mercer. A real-time monitor for a distributed real-time operating system. *ACM SIGPLAN Notices*, 24(1):68–77, January 1989.

[15] L. D. Wittie. Debugging distributed C programs by real time replay. *ACM SIGPLAN Notices*, 24(1):57–67, January 1989.

Colin Fidge

Colin Fidge is a Senior Research Fellow with the Software Verification Research Centre, Department of Computer Science, University of Queensland. He completed his Ph.D. at the Australian National University. His research interests include formal development methods for concurrent and real-time systems. He is currently coordinator of the *Quartz* research project, producing a formal development method for real-time, multi-tasking Ada 95 programs.