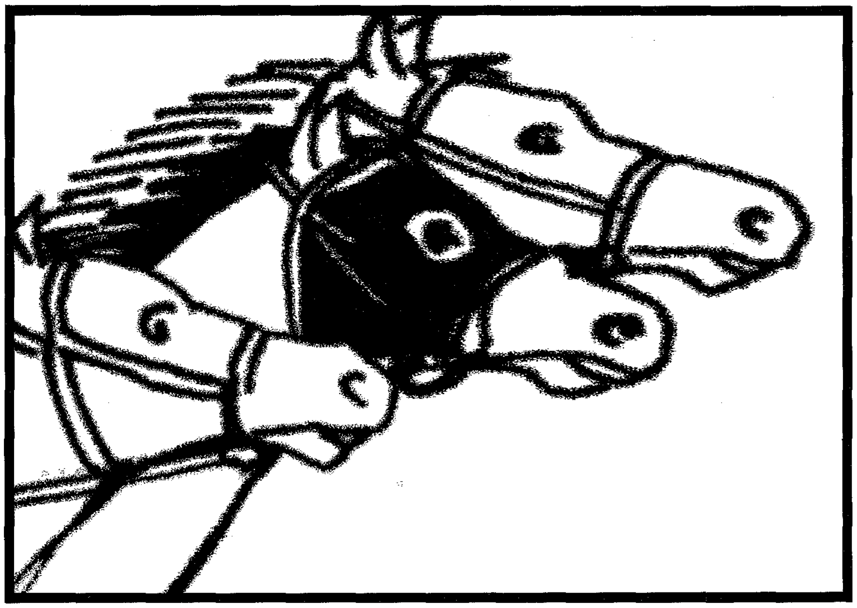
Fundamentals of Distributed System Observation

It's difficult to determine event order in distributed systems because of the observability problem. The author discusses this problem and evaluates different strategies for determining arrival order.



COLIN FIDGE
University of Queensland

Testing and debugging a distributed system presents the programmer with profound challenges: merely observing what is happening in a network of processes is difficult. This observability problem makes it hard to accurately determine event order during a given computation.

An obvious approach to determining event order is to rely on the arrival order of notification messages, which tell the programmer that an event has occurred. If a notification of event *e* arrived before a notification of event *f*, you might assume *e* executed first. However, there are a number of influences on observability that make arrival order insufficient for judging the actual order in which events occur. These include unpredictable processor preemptions, variable message transmission delays, and different communication times depending on the observer's location.

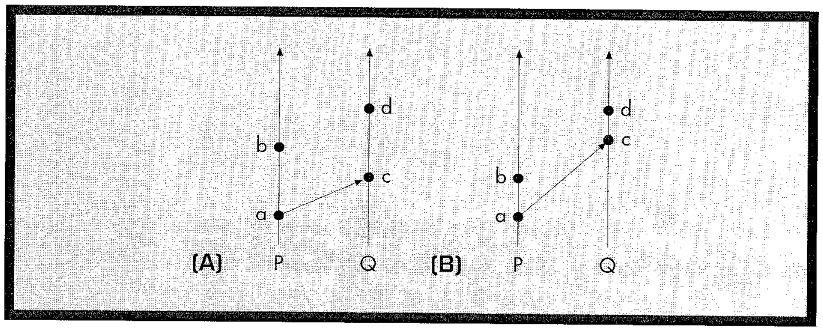


Figure 1. Two views of a single distributed computation.

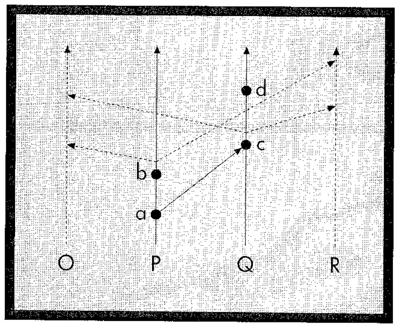


Figure 2. Multiple observers can see different event orderings.

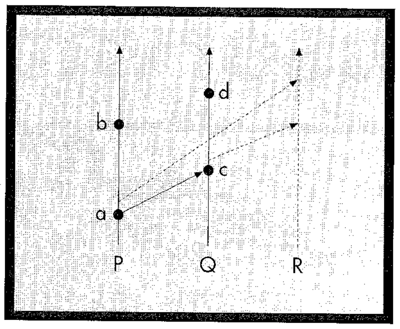


Figure 3. Incorrect perception of event ordering can be caused by delayed notifications.

For these reasons, programmers sometimes rely upon time stamping: a number is assigned to each event and included within the notification message. The observer can then use these values to determine the true order in which events occurred.

I have analyzed four time-stamping methods to determine their effectiveness in contending with observability problems. Although my work focuses on distributed systems, the concepts also apply to any system exhibiting concurrency—the appearance of two or more events occurring simultaneously—including multiprocessor machines and uniprocessor multitasking. Events in this context may be the execution of single machine instructions or entire procedures; the level of granularity is unimportant. To define event order, I use the idea of causality—the ability of one event to affect another—because it allows us to reason independent of any particular time frame.[1]

I also 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 first in, first out queuing is not necessarily assumed). However, the concepts I discuss extend directly to synchronous message-passing, shared memory, remote procedure calls or rendezvous, and so on.

OBSERVABILITY PROBLEMS

An *observer* is any entity—a person or a network process—that attempts to examine a computation. Observers may watch the system while the computation is in progress or examine a postmortem event log or trace. In either case, the observer must be informed when interesting events occur; typically, system probes either send notification messages to the observer or write entries into the log when an event occurs.

Imagine, for example, that you are one of the observers of the distributed system in Figure 1a. The computation is simple: The system consists of two parallel processes *P* and *Q*. Process *P* performs two events: event *a* denotes message transmission; *P* then performs *b*, an action local to itself. Process *Q* also performs two events, *c* and *d*, *c* being the reception of the message sent by *P*.

As Figure 1b shows, this same computation can be redrawn so that the same events occur in the same relative local orders. Only the omnipotent viewpoint provided by such diagrams can tell us that independent events *b* and *c* are interleaved differently; processes *P* and *Q* cannot "see" any difference.[1] Clearly, correctly ordering such events is problematic. When you rely solely on the arrival order of notification messages to determine event orderings in a distributed system, four types of discrepancies can arise.

Multiple observers, different orderings. Observers of a particular computation may perceive different event orderings. Observers *O* and *R* in Figure 2 are notified that events *b* and *c* have occurred (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* believes *c* occurred before event *b*. Both interpretations are valid, but they cannot be easily reconciled. (There is an obvious parallel with space-time physics: the observer's location determines its view of the universe.)

Incorrect perceived orderings. More problematic is when the event ordering you perceive is simply incorrect, as Figure 3 shows. This error can be caused by notifications that are delayed due to retries or routing to the observer through indirect pathways.

Same computation, different orderings. When testing or debugging a system, programmers typically replay the same computation several times so they can study different aspects of its behavior. Unfortunately, they may see different event orderings with each replay!

Figure 4 shows two different instances of the same computation. In

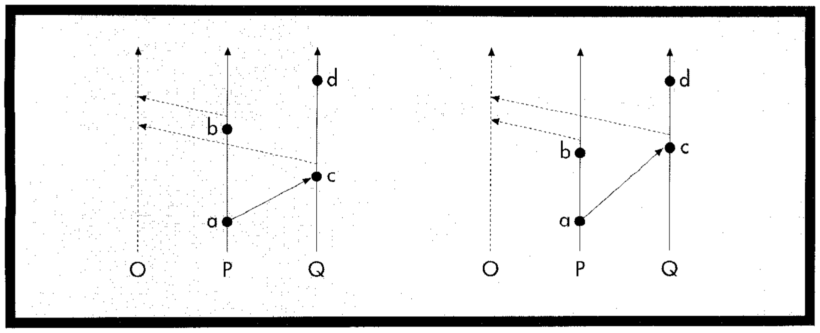


Figure 4. The same computation can exhibit different event orderings.

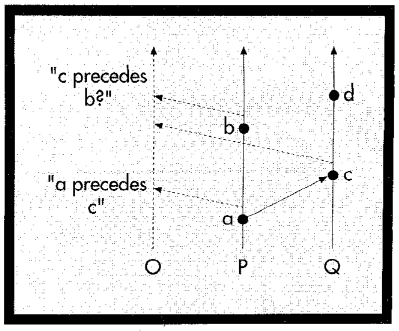


Figure 5. Relying upon notification arrival time can lead you to assume arbitrary event orderings.

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 behavior during debugging may be due to minor differences in the processor and link loads caused by other system activity and can occur even with a deterministic computation! This can be a major source of frustration; even though you're using a replay mechanism, ordering errors you observed previously can simply vanish.

Arbitrary orderings. When you rely on notification arrival time to determine event orderings you may assume arbitrary orderings between unrelated events. In Figure 5, observer O first sees that event a occurred before event c. This is a valid observation: a *must* occur before c in this computation. Observer O then sees event c occur before event b. This perceived ordering is merely an artifact of the notification mechanism. If you compare Figures 1a and b, for example, you can see that events b and c are independent and can 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 your 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.

TABLE 1
EFFECTIVENESS OF TIME-STAMPING MECHANISMS

	Ordering Mechanisms				
	Real-time time stamps			**Logical time stamps**	
Effects	**arrival ordering**	**local clocks**	**global clock**	**totally ordered**	**partially ordered**
Multiple observers see different orderings		✓	✓	✓	✓
Incorrectly perceived orderings			✓	✓	✓
Same computation exhibits different orderings				✓	✓
Arbitrary orderings adopted					✓

EFFECTIVENESS OF TIME STAMPING

Table 1 summarizes my analysis of four different time-stamping mechanisms; a check indicates that the mechanism successfully overcomes a particular observability problem.

Local real-time clocks. One obvious source for time stamps is to use whatever real-time clock is available in each processor's hardware. 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 6 shows two possible ways to time-stamp the events in the sample computation. Because the clocks on different processors are not synchronized, they will inevitably drift. Incorrect orderings result: on the left, 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 can receive different time stamps, as the scenarios in Figure 6 show. Finally, the ordering between independent events, such as b and d, is randomly influenced by processor loads and the inability of the clocks to remain synchronized.

Global real-time clocks. Let's assume that the clocks are synchronized throughout the distributed system to a high degree of accuracy, in effect providing a global reference for real time. This avoids incorrect orderings, and time readings become meaningful across processor boundaries and hence always reflect the actual order of event occurrence.

Nevertheless, as Figure 7 shows, the same computation can still yield different orderings if system loads vary between tests, and independent events are still arbitrarily ordered.

It is perhaps surprising that such a

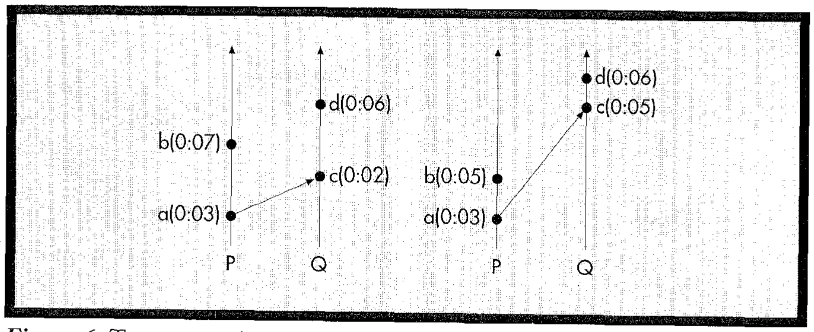


Figure 6. Two ways to time-stamp events using unsynchronized local clocks.

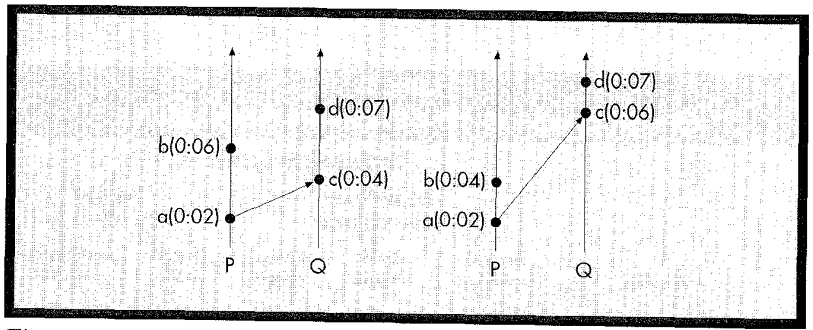


Figure 7. Real-time time stamps use a global clock to provide synchronization. Still, different orders can result.

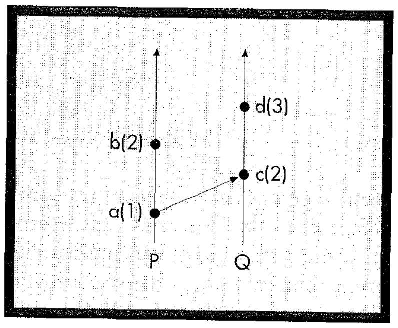


Figure 8. Logical time stamps using a totally ordered logical clock.

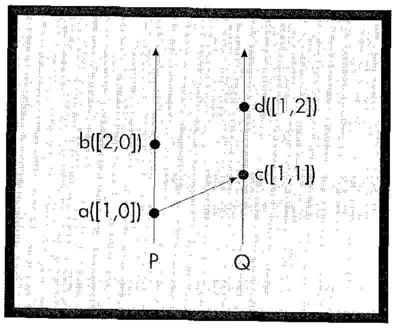


Figure 9. Logical time stamps using a partially ordered logical clock.

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 would require an unbounded number of tests!

Totally ordered logical clocks. The outstanding problems described so far are tied to using *absolute* time to order events. These values are randomly influenced by factors such as processor loads and the start time of each process. *Logical* clocks have a more objective ordering mechanism and thus provide a possible solution. (As the box on page 81 describes, certain closed-world conditions must be met for their use).

A simple system of logical clocks can be used to totally order distributed-system events using the following rules[1]:

- ♦ Each process maintains an integer counter.
- ♦ When a process performs an event of interest, the counter value increases.
- ♦ When a process sends a message, the current counter value is "piggybacked" on the message.
- ♦ When 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 8 shows the totally ordered time stamps associated with each event, 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 time stamp 2, rather than 1, because it must have a higher value than the corresponding send event. The time stamps thus generated are not unique, as events *b* and *c* show. The total ordering is completed by adopting an arbitrary but consistent ordering among processes when two events have the same time stamp.[1]

This mechanism has the same advantages as global real-time clocks and also precludes the possibility of the same computation producing different orderings. The time stamps are consistently associated with each event regardless of the number of replays or differences in absolute timing (assuming the algorithm for increasing time stamps is deterministic). This consistency is an important advantage during testing and debugging because it allows you to avoid repeating a computation to see if different orderings are observable (a nondeterministic program may still generate several different *computations*, however).

The advantages of totally ordered logical clocks, along with their ease of implementation, have led to their use in many distributed debugging systems. However, one issue remains: arbitrary ordering is still imposed on independent events. An observer relying on the time stamps in Figure 8 will mistakenly conclude that *b* always occurs before *d*, even though no interaction between processes *P* and *Q* guarantees this. This misleading view will thwart any attempts to identify problems stemming from inadequate synchronization between events.

Partially ordered logical clocks. The ordering of events defined by totally ordered clocks is an incomplete view of event causality. However, with a straightforward extension, causal orderings can be preserved. Partially ordered logical clocks have been identified by several researchers as a way to record the whole causality relationship.[2-4] The clocks do this by treating each time stamp not as a single number, but as an array, or vector, of numbers.

Partially ordered logical clocks can be maintained as follows[2,4]:

♦ Each process maintains an array of counters, with one element in the array for each distributed-system process.

♦ When a process performs an event of interest, it increases the counter value associated with this process in its array.

♦ When a process sends a message, the array of counters is piggybacked on the message.

♦ When a process receives a message, it sets each element in its array to equal either its old value or the value of the corresponding element in the piggybacked array, whichever is larger.

Figure 9 shows how our example would be time-stamped. Processes P and Q both maintain an array of two counters: the first counter value represents the number of events known to have occurred in process P; the second, 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.[2])

The entire array forms the time stamp. When comparing two such time stamps, you can conclude that some event e, occurring in process i, preceded some event f, occurring in process j, only if event f's time stamp has a counter value for process i greater than or equal to the counter for process i in event e's time stamp, and event e's time stamp has a counter value for process j strictly less than that for process j in event f's time stamp.

The second condition avoids reflexivity[1] and allows for the case of synchronous message passing, where corresponding send and receive events are treated like a single action and receive the same time stamp. A simpler test is available if only asynchronous message passing will be used.[4]

For example, in Figure 9 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 knows of no events in process Q, whereas d knows of 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. You cannot show that b precedes d because b knows of more events (2) in process P than does d (1). Furthermore, you 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 use these time stamps 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.

Thus, the remaining observability problem is resolved; partially ordered clocks reflect only true causal orderings and make the *absence* of ordering explicit.

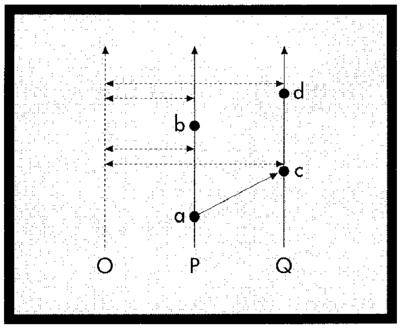


Figure 10. Inaccurate reporting can result from synchronous notifications.

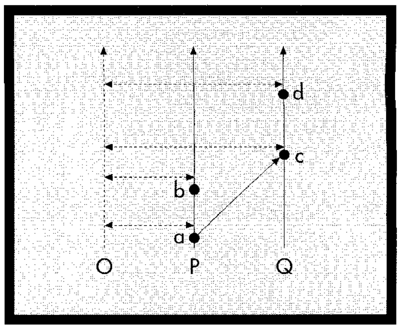


Figure 11. Intrusiveness due to synchronous notifications.

A CLOSED-WORLD ASSUMPTION

To accurately model event relationships in the logical time-stamping mechanisms I discuss here, two conditions are essential.

♦ All system processes must participate in the time-stamping algorithm.

♦ All interactions between processes must be time stamped.[1]

If any process does not propagate time stamps correctly, or if the processes can interact through some covert channel that is not time stamped (such as a file system), then the clock values may not accurately reflect causal relationships.[2]

Conversely, a passive observer process must *not* participate in the time-stamping algorithm if it is to be unintrusive. If the observer propagates time stamps 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.

REFERENCES

1. D.R. Cheriton and D. Skeen, "Understanding the Limitations of Causally and Totally Ordered Communications," *Operating Systems Rev.*, Dec. 1993, pp. 44–57.
2. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, July 1978, pp. 558–565.

SYNCHRONOUS NOTIFICATIONS

Many observability problems stem from unpredictable delays between the time that distributed-system events occur and the time that an observer is notified. It is therefore tempting to assume that using synchronous communication between the system and its observers will avoid these effects. Unfortunately, as Figure 10 shows, the

THE PROBE EFFECT

The probe effect (sometimes referred to as the "Heisenberg effect" by aspiring physicists) is often associated with, but distinct from, the observability problem. The probe effect occurs when a debugger adds auxiliary code that alters the behavior of a concurrent program.[1] 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.

Avoidance behavior. Many systems programmers take extreme measures in an attempt to avoid the probe effect, typically by trying to account for the time occupied by the auxiliary code.[2,3] Unfortunately, software-based debugging utilities inevitably introduce some degree of intrusiveness.[4] (Customized hardware can be used to unintrusively monitor a system,[5] 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,[1,5-7] albeit with a penalty in terms of runtime overheads. (This approach has the benefit of leaving debugging "hooks" in an operational system to track infrequent errors that eluded testing and debugging, but such access points can also be a security hazard!)

Reproducibility. We must clearly distinguish the probe effect from the difficulty of achieving reproducibility while observing concurrent software. Having seen the system perform some behavior 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.[5] These include recording traces for later replay[1] or giving the programmer explicit control over nondeterministic alternatives.[8]

Practical debugging problems attributed to the probe effect are, quite often, actually manifestations of the difficulty of achieving reproducibility.

REFERENCES

1. P.S. Dodd and C.V. Ravishankar, "Monitoring and Debugging Distributed Real-Time Programs," *Software—Practice & Experience*, Oct. 1992, pp. 863–877.
2. F. Baiardi, N. de Francesco, and G. Vaglini, "Development of a Debugger for a Concurrent Language," *IEEE Trans. Software Eng.*, Apr. 1986, pp. 547–553.
3. L.D. Wittie, "Debugging Distributed C Programs by Real-Time Replay," *ACM SIGPLAN Notices*, Jan. 1989, pp. 57-67.
4. H. Tokuda, M. Kotera, and C. Mercer, "A Real-Time Monitor for a Distributed Real-Time Operating System," *ACM SIGPLAN Notices*, Jan. 1989, pp. 68-77.
5. C.E. McDowell and D.P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surveys*, Dec. 1989, pp. 593–622.
6. D. Haban, "DTM: A Method for Testing Distributed Systems," *Proc. 6th Symp. Reliability in Distributed Software and Database Systems*, IEEE CS Press, Los Alamitos, Calif., 1987, pp. 66-73.
7. A. Gordon, *Ordering Errors in Distributed Programs*, doctoral dissertation, Univ. Wisconsin-Madison, Madison, Wisc., 1985.
8. J. Joyce et al., "Monitoring Distributed Systems," *ACM Trans. Computer Systems*, Feb. 1987, pp. 121–150.

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.

Inaccurate reporting. It is still possible for the notification arrival time to reflect event orderings incorrectly. In Figure 10, 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, such as between b and d. Similarly, multiple observers may see different orderings and a single computation may yield different observations.

Intrusive observers. Synchronous notification messages cannot solve our observability problems. Even worse, synchronous notification introduces a form of *probe effect*, in which the mere act of observing the system alters its behavior—again there is a parallel with quantum physics. (The box, "The Probe Effect," above provides more detail.)

The probe effect manifests in two ways. First, processes that wish to notify the observer are effectively blocked until the observer deigns to communicate with them. This can alter real-time behavior and nondeterministic choices in the system. Also, any bias on the observer's part about "preferred" system processes to communicate with will influence the observed processes' ability to proceed.

Second, the bidirectional causality relationship defined by synchronous communication creates new causal orderings that would not exist in the observer's absence. If you follow the arrows from b to d in Figure 11, you can see how this occurs. Each event in the figure 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, which means that event b potentially causally affects event d!

Although in my study partially ordered logical clocks proved to be the only time-stamping method that fully indicates ordering between events, they are not the only mechanism you should use because

- ♦ they are notoriously expensive to implement,[5] and
- ♦ the array size must be as great as the number of parallel processes.[6]

Many schemes have been suggested for

reducing their cost, but all either involve some loss of causality information or merely trade storage requirements for communication and processing overheads.

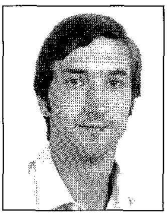
Nevertheless, when you use other time-stamping mechanisms, you should appreciate their limitations and understand that they offer an incomplete view of event ordering. Partially ordered clocks can then be used to give the complete picture when necessary.

ACKNOWLEDGMENTS

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

REFERENCES

1. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, July 1978, pp. 558–565.
2. C.J. Fidge, "Logical Time in Distributed Computing Systems," *Computer*, Aug. 1991, pp. 28–33.
3. M. Raynal and M. Singhal, "Capturing Causality in Distributed Systems," *Computer*, Feb. 1996, pp. 49–56.
4. F. Mattern, "Virtual Time and Global States Distributed Systems," *Parallel and Distributed Algorithms*, M. Cosnard et al., eds., North-Holland, Amsterdam, 1989, pp. 215–226.
5. D.R. Cheriton and D. Skeen, "Understanding the Limitations of Causally and Totally Ordered Communications," *Operating Systems Rev.*, Dec. 1993, pp. 44–57.
6. B. Charron-Bost, "Concerning the Size of Clocks," in *Semantics of Systems of Concurrent Processes*, I. Guessarian, ed., Vol. 469 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1990, pp. 176–184.



Colin Fidge is a senior research fellow with the Software Verification Research Centre, Department of Computer Science, University of Queensland. His research interests include formal development methods for concurrent and real-time systems. He is coordinator of the Quartz research project, producing a formal development method for real-time, multitasking Ada 95 programs.

Fidge received a PhD in computer science from the Australian National University in 1990.

Address questions about this article to Fidge at the Software Verification Research Centre, Department of Computer Science, The University of Queensland, Queensland 4072, Australia; phone, 61-7-3365-1648; fax, 61-7-3365-1533; cjf@cs.uq.edu.au.