SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 93-15

Fundamentals of Distributed System Observation (version 1.1)

Colin Fidge

November 1993

Phone: +61 7 365 1003
Fax: +61 7 365 1533

للاستشارات المنارة

www.manaraa.com

Fundamentals of Distributed System Observation

Colin Fidge
Software Verification Research Centre
Department of Computer Science
The University of Queensland
Queensland 4072, Australia

Abstract

Fundamental issues associated with observing activity in a distributed system are identified. The limitations of various timestamping mechanisms used in testing and debugging such systems are described.

1 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. Herein we characterise the issues and then use this classification to critically analyse the power of timestamp-based solutions, thus giving an insight into the expressibility of different time models.

2 Definitions

The effects discussed herein usually manifest themselves in *distributed* systems, *i.e.*, those in which there are a number of communicating, spatially-separated processors. However, the concepts discussed below are applicable to any system exhibiting *concurrency*, *i.e.*, the appearance of two or more events occurring simultaneously, including multiprocessor machines and single processor interleaved tasking.

We make no assumptions about the nature of *events* occurring in the system other than to note that they represent discrete actions meaningful to the programmer. This may be the execution of single machine instructions, or entire procedures; the level of granularity is irrelevant.

For brevity in this presentation we 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, *e.g.*, synchronous message-passing, shared memory, remote procedure calls or rendezvous.

Our motivating concern is the *observability problem* in distributed systems, *i.e.*, 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 via the particular set of control paths followed on this occasion). We adopt *causality*, *i.e.*, 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.

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. Observers may watch the system "live", while the computation is in progress, or examine a post-mortem event log or trace. In any case it is necessary to inform the observer whenever interesting events occur. In practice this involves instrumenting the system under obser-

www.manaraa.com

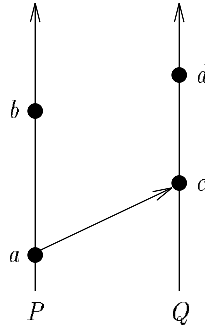

Figure 1: A distributed computation.

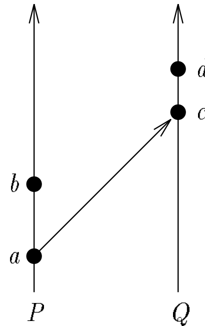

Figure 2: A different view of the same computation.

vation with *probes* that send *notification* messages to the observer (or, equivalently, write entries into the log) following the occurrence of such events.

Our concern in this paper is to determine how accurate a view of system activity is presented to the observer by these notifications. We assess a number of proposed methods of event *timestamping*, *i.e.*, ways of associating timing information 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, we use the computation shown in Figure 1. It shows a system consisting of two parallel processes P and Q. Process P performs two events. Firstly, event a denotes the transmission of a message and 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.

It is important to note that, in the absence of any further information, this same computation can be redrawn as shown in Figure 2. Here the same events occur in the same relative *local* orders, but our "omnipotent" viewpoint allows us to interleave independent events such as b and c differently. Lamport [1] discusses this equivalence in depth.

3 Fundamental observability effects

The effects an observer may encounter when relying on the arrival of notification messages to determine event orderings in a distributed system can be summarised in the following four ways.

3.1 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 3 two observers O and R are notified of the occurrence of events b and c (notification messages are shown as dotted arrows). Due to the propagation 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. (We can draw an obvious parallel with spacetime physics—the observer's location determines its view of the universe [2].)

3.2 Incorrect perceived orderings

More seriously, the perceived event ordering may simply be incorrect. In Figure 4 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. (This does not happen in theoretical physics because we rely on the constant

المنارة للاستشارات
www.manaraa.com

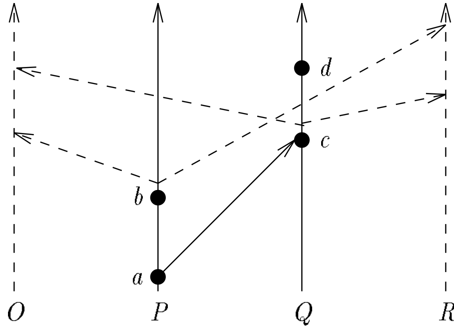

Figure 3: Multiple observers see different orderings.

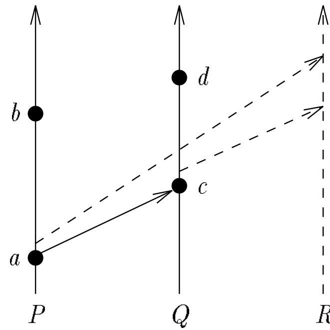

Figure 4: Incorrect perceived ordering.

upper bound placed on information propagation due to the speed of light.)

3.3 Same computation exhibits different orderings

When testing or debugging a system we frequently want 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 5 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 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.

3.4 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 6 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 artifact of the notification mechanism. As shown by comparing Figures 1 and 2, 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 ordering.

4 Effectiveness of time-stamping

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.

المنارة للاستشارات
www.manaraa.com

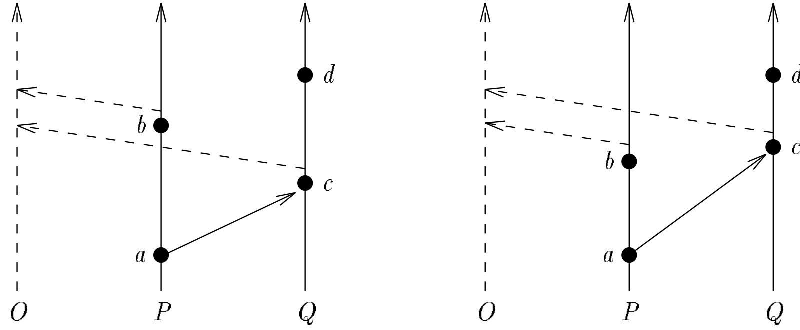

Figure 5: Same computation exhibits different orderings.

	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.

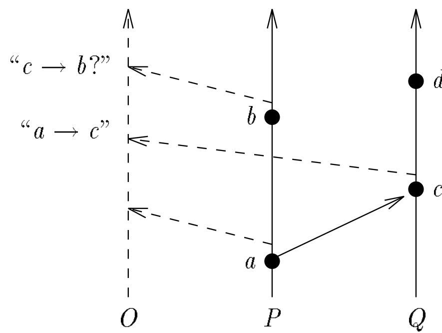

Figure 6: Arbitrary orderings adopted.

The remainder of this section critically analyses the ability of four different timestamping mechanisms to resolve the observability effects described in Section 3 above. Table 1 summarises the results; a '✓' indicates that the proposed timestamping mechanism satisfactorily overcomes the effect process distribution has on observability.

4.1 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. Since all notification messages will then have the same time value associated with each event this means that all observers will see the same time orderings, thus avoiding the first effect. Unfortunately the others persist. Figure 7 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

المنارة للاستشارات
www.manaraa.com

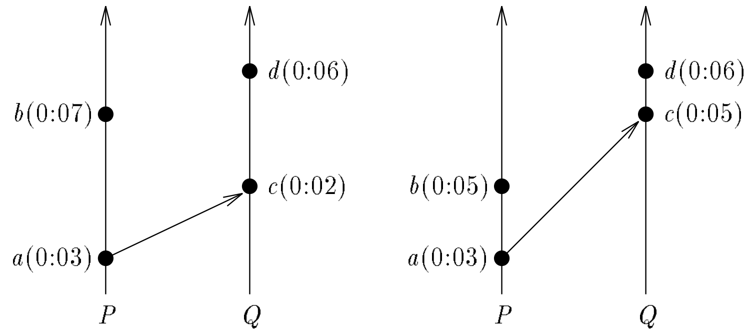

Figure 7: Real-time timestamps using unsynchronised local clocks.

inevitably drift, so it is not meaningful to compare times across machine boundaries. Incorrect orderings may therefore be seen; on the left of Figure 7 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 7. Finally, the ordering between independent events, such as b and d in Figure 7, is randomly influenced by processor loads and the (in)ability of the clocks to remain synchronised.

4.2 A global real-time clock

As an improvement assume that the clocks are synchronised throughout the distributed system to a high degree of accuracy, in effect giving us 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 8, the same computation may still yield different orderings, and arbitrary orderings are still imposed on independent events.

It is surprising that such a powerful facility as global real time, which is not realisable to sufficient accuracy in practice, 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!

4.3 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 absolute 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 [1] using the following rules:

- each process maintains an integer counter,
- whenever a process performs an event of interest it increments its counter,
- 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 9 shows the totally ordered timestamps associated with each event in our example computation, assuming that the counters start from zero and are incremented by exactly 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 to-

المنارة للاستشارات
www.manaraa.com

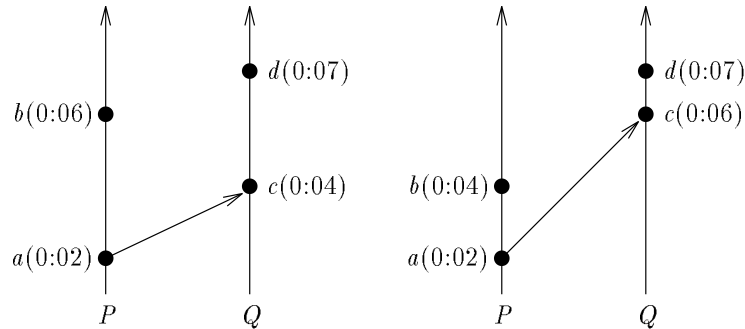

Figure 8: Real-time timestamps using a global clock.

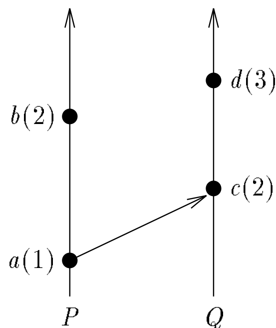

Figure 9: Totally ordered logical clock timestamps.

tal ordering is completed by adopting an arbitrary, but consistent, ordering among the processes when two events have the same timestamp.

This mechanism has the same advantages of 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 the absolute timing. 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 9 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 debug problems stemming from inadequate synchronisation between events.

4.4 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 [3, 4] can be maintained as follows:

- 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 increments 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.

المنارة للاستشارات
www.manaraa.com

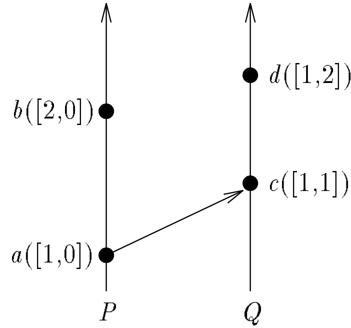

Figure 10: Partially ordered logical clock timestamps.

Figure 10 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 assumes a fixed number of processes, but the concept extends to dynamic process creation [3].)

The entire array forms the timestamp. When comparing two such timestamps we can conclude that some event e preceded some event f if and only if

- event f has a counter value for the process in which e occurred greater than or equal to the number of events in that process up to e inclusive, and
- event e has a counter value for the process in which f occurred strictly less than the number of events in that process up to f inclusive (this condition avoids reflexivity [1] and allows for synchronous message-passing [3]).

For instance, in Figure 10 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 in process Q as d. Similarly we know that c preceded d because d knows of more events in process Q (2) than c does (1).

These conclusions can also be reached 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 d does (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 "enforced" causal orderings and make the *absence* of ordering explicit.

5 Discussion

5.1 Synchronous notifications

Many of the observability effects defined in Section 3 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 11, 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; a synchronous message-pass can be modelled as an asynchronous message sent to the observer immediately followed by an acknowledgement message sent back to the notifier.)

It is still possible for the arrival time of notifications to incorrectly reflect event orderings. In Figure 11 process P is delayed, perhaps due to contention for the CPU, and the notification for event c arrives before that of its causal predecessor a. 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.

Making the notifications an integral part of

المنارة للاستشارات
www.manaraa.com

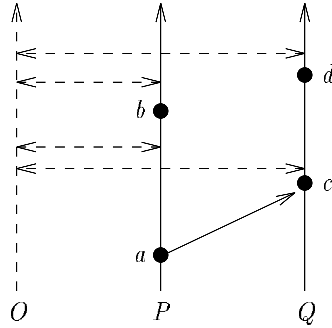

Figure 11: Inaccurate reporting using synchronous notifications.

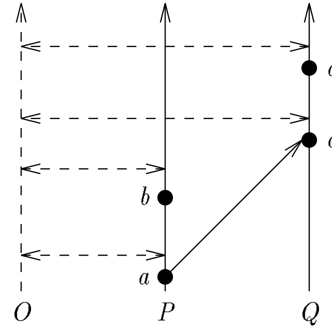

Figure 12: Intrusiveness due to synchronous notifications.

the events themselves, *i.e.*, making the event and the notification atomic, improves the situation but this is difficult to achieve in practice. In a geographically-separated network, synchronous message-passing is inevitably implemented via an underlying *asynchronous* message protocol.

5.2 Intrusive observers

It was noted above that synchronous notification messages cannot solve our observability problems. Even worse, synchronous notification introduces a form of *probe effect* (see Appendix A) in which the mere act of observing the system alters its behaviour! (Again one can draw 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 the observer. In Figure 12 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 direct causal link between P and Q, via the observer O, that means that event b causally affects event d! This is easily demonstrable by following arrows from b to d in Figure 12.

(A similar problem happens whenever an observer participates in the logical time-stamping algorithms described in Sections 4.3 and 4.4. It is important that an observer must not propagate the timestamps it receives if it is to remain unintrusive. Lamport also discusses the opposite problem, in which the failure of some links in a distributed system to propagate timestamps leads to "anomalous" observations [1].)

6 Conclusion

We have compared a number of mechanisms intended to give the programmer a view of the order in which events occur in a distributed system. It was shown that partially ordered 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 clocks are the only mechanism that should be used; they are expensive to implement and totally ordered logical clocks have demonstrated their adequacy in many practical applications. However programmers using other timestamping mechanisms should appreciate their limitations and understand that they are receiving an incomplete view of event

المنارة للاستشارات
www.manaraa.com

ordering. Partially ordered clocks can then be used to give the complete picture when needed.

This work began 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 central "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 available at the time (1985) was capable of giving us 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 to the development of partially ordered logical clocks [3] as a model that can detect the *absence* of ordering.

Acknowledgements I wish to thank Mark Utting and the anonymous referees for their insightful comments and corrections, and Robin Stanton and Amr El-Kadi for influential discussions on the nature of the probe effect.

References

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

[2] E.F. Taylor and J.A. Wheeler. *Spacetime Physics: An Introduction to Special Relativity*. Freeman, 1992. Second edition.

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

[4] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[5] J. Gait. A probe effect in concurrent programs. *Software—Practice & Experience*, 16(3):225–233, 1986.

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

[7] 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.

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

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

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

[11] R. Rubin, L. Rudoplh, and D. Zernik. Debugging parallel programs in parallel. *ACM SIGPLAN Notices*, 24(1), January 1989.

[12] D. Haban. DTM: A method for testing distributed systems. In *Proc. Sixth Symposium on Reliability in Distributed Software and Database Systems*, Virginia, March 1987.

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

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

المنارة للاستشارات
www.manaraa.com

A The probe effect and reproducibility

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, 6]. Whereas the observability problem concerned our 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 can 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 [7, 8]. Unfortunately it has long been recognised that software-based debugging utilities inevitably introduce some degree of intrusiveness [9]. (Customised hardware can be used to unintrusively monitor a system [10, 11] 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.

We can therefore correspondingly recognise that the probe effect may be lessened under certain conditions. In contemporary programming languages (*e.g.*, Ada and occam) it is generally safe to assume that the language semantics makes no commitment regarding fairness of choices so no amount of bias introduced by debugging code can be said to have altered the program semantics.

Given this assumption, there are two particular scenarios of interest. Firstly, it has been widely recognised that if the program undergoing debugging is the *same* as the final 'production' version then the probe effect is not a concern. This implies the commonly-suggested solution of permanently installing debugging probes so that there is only ever one version of the program in existence [10, 6, 12, 13], albeit with a penalty in terms of run-time overheads. (However, 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.)

Secondly, if the program has no real-time deadlines and does not attempt unprotected access to shared memory, then we can be certain that no *new* logical errors will be induced in the augmented program—only valid control paths can be followed. However there is still the possibility that existing errors are hidden, due to the probability of an erroneous computation occurring being reduced to near zero.

It is also important to clearly distinguish the probe effect from the difficulty of achieving *reproducibility* while debugging concurrent software. This is the problem that, having seen the system perform some behaviour of interest, the programmer cannot force this particular computation to occur again. However the reproducibility problem 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 [10]. These include recording traces for later replay [6] or giving the programmer explicit control over nondeterministic alternatives [14]. (Interestingly, the effect described in Section 3.3 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.

المنارة للاستشارات
www.manaraa.com