Logical Time in Distributed Computing Systems

Colin Fidge, University of Queensland

Partially ordered logical clocks can provide a decentralized definition of time for distributed computing systems, which lack a common time base.

Unlike conventional sequential programs, the computations performed by distributed computing systems do not yield a linear sequence of events. The interrelationships between the events performed in a distributed system inherently define a partial ordering — genuinely concurrent events have no influence on one another.

In the past, designers have typically used a simplified view of distributed computations, imposing an interleaved total ordering on the events performed. However, new computing concepts now let us use the full partial ordering of events as defined by their causal relationships, that is, the ability of one event to directly, or transitively, affect another. In this article I define this partial ordering, describe its generalized and practical implementations in terms of partially ordered logical clocks, and summarize some current applications of the new technology.

Definition

For a system using both asynchronous and synchronous message-passing and process-nesting, the causal relationships between events in a distributed computation are captured as follows.

The relation "happened before," denoted $\rightarrow$, is the smallest relation satisfying the six conditions listed below. It is an irreflexive, transitive relation among the *events* performed during a given *computation*. An event is a uniquely identified runtime instance of an atomic action of interest, and a computation is a particular run or execution of the distributed computing system. A computation consists of one or more possibly nested *process instances*, which are uniquely identified runtime instantiations of a particular process definition.

- *Condition 1: Sequential behavior.* If events e and f occur in the same process instance p, and f occurs after e, then $e \rightarrow f$.
- *Condition 2: Process creation.* If event e and process instance q occur in process instance p, event f occurs in q, and q begins after e, then $e \rightarrow f$.
- *Condition 3: Process termination.* If event e and process instance q occur in process instance p, event f occurs in q, and e occurs after q terminates, then $f \rightarrow e$.

0018-9162/91/0800-0028$01.00 © 1991 IEEE

• *Condition 4: Synchronous (unbuffered) message-passing.* If event e is a synchronous input (output) and event f is the corresponding output (input), and there is an event g such that $e \rightarrow g$, then $f \rightarrow g$. If there is an event h such that $h \rightarrow e$, then $h \rightarrow f$.

• *Condition 5: Asynchronous (buffered) message-passing.* If event e is an asynchronous send, and event f is the corresponding receive, then $e \rightarrow f$.

• *Condition 6: Transitivity.* If $e \rightarrow f$ and $f \rightarrow g$, then $e \rightarrow g$.

An event e "occurs in" a process instance p if p executes e. A process instance q occurs in a process instance p if q is a subprocess of p. "After" is used only when referring to actions that occur within a single process instance. Each process consists of a *sequence* of actions.[1]

Partially ordered logical clocks

Figure 1 shows a computation performed by a distributed program P. Shortly after it starts executing, P spawns two process instances Q and R. All three processes perform a number of events. Some are actions internal to a process, such as E, or are communication actions such as G. (For clarity I treat process creation and termination as special cases, although it is possible to consider these actions as synchronizing events.) After performing event H, process R creates two processes S and T, and suspends itself during the lifetime of its offspring. Processes P and S communicate via synchronous message-passing; event F outputs a message that is simultaneously input by event I. Processes P and Q communicate via asynchronous message-passing; event G denotes the sending of a message later received by event C.

After processes S and T terminate, process R resumes execution and performs event L. After processes Q and R terminate, the main program, process P, performs a final event M, and the computation ends.

Partially ordered logical clocks characterize all interrelationships between these events. In Figure 1 the annotation at each event shows the partially ordered time at that point in the computation. I explain the maintenance and use of this information below.

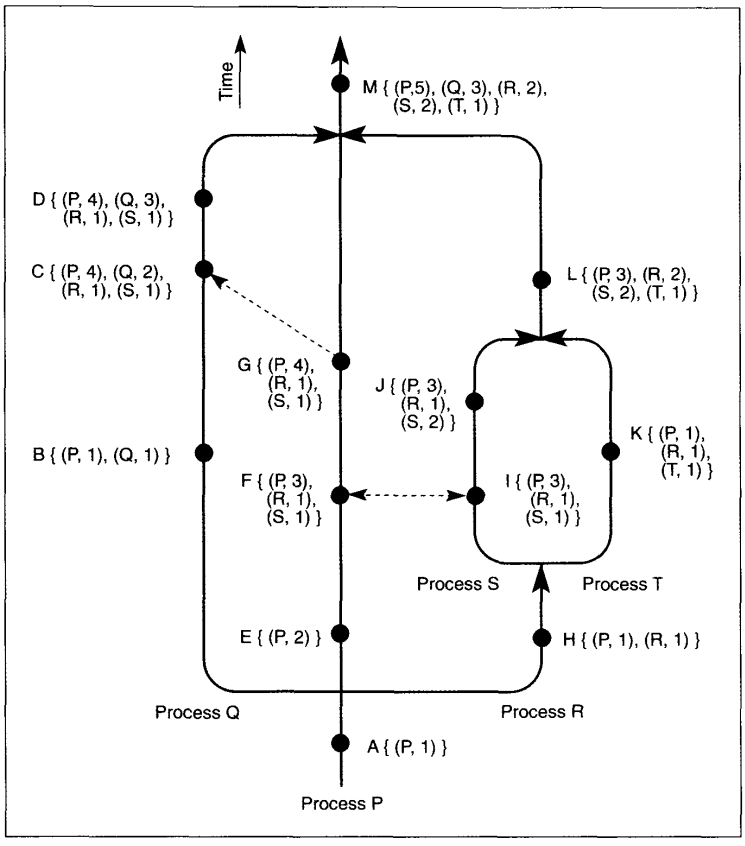

Figure 1. A distributed computation. Expressions in braces show the partially ordered time at each event. Solid arrows represent flow control; dotted arrows show interprocess communication.

Notation. Because the time readings must be partially ordered, a single integer or real value is inadequate as a data structure. Assume that p_i denotes a process instance uniquely identified by i. We can represent a partially ordered time reading made by p_i with a set of pairs

$$\{(i,n_i),\ldots,(l,n_l)\}$$

Here each pair consists of a process instance identifier j and a numerical "counter" value n_j representing the value of the counter in p_j as perceived by p_i. Each process thus maintains knowledge of the counters in all other processes of which it has heard. Process instances not represented in the set have the default counter value 0.

Assume that e_i represents a unique event e performed by process instance p_i, and t_{e_i} is the time stamp attached to some permanent record of the execution of this event. Then $t_{e_i}(j)$ is the value of the counter for p_j in this time stamp.

For example, in Figure 1,

$$t_G = \{(P,4), (R,1), (S,1)\}$$

is the time when process P performed event G. (Because event names are unique in Figure 1, I omit the process identifier subscripts: G is equivalent to G_P.) From this we can determine the "local" counter value for process P at that time,

$$t_G(P) = 4$$

and the last known value for process S,

$$t_G(S) = 1$$

When it performed event G, process P had received no information from process Q, hence

$$t_G(Q) = 0$$

A handy function in the following definitions is *max*. Given one or more partially ordered time readings, *max* returns a time reading in which every counter is set to the maximum of all corresponding values in the arguments presented to it, for example,

$$max(\{(i,2), (j,1), (k,3)\}, \{(i,4), (k,1)\}) = \{(i,4), (j,1), (k,3)\}$$

The counter for p_j has the default value 0 in the second argument to *max* above.

Rules. A computation can maintain a partially ordered logical clock using nine rules.[2] For the rules below to apply, each process instance p_i created during the lifetime of the computation (including the outer-level main program) must maintain an auxiliary variable c_i to hold the current partially ordered time as perceived by p_i.

• *Rule A: Initialization.* When the program begins execution, the time is initialized to the empty set, that is, $c_m := \{\}$, where m is the process instance identifier associated with the main program.

Since zero-valued counters are implicit, {} is equivalent to the infinite set $\{(i,0), (j,0),...\}$ for every possible process identifier.

• *Rule B: Ticking.* Whenever a process instance p_i performs an event, it increments $c_i(i)$ at least once.

For instance, in performing event D, process Q increments its own counter from 2 to 3.

In Figure 1 each counter has been incremented exactly once for each event performed. However, the rules are valid for any number of "ticks," as long as the counters never decrease.

• *Rule C: Monotonically increasing counters.* No counter in any c_i is ever decremented.

• *Rule D: Process creation.* Whenever a process instance p_i creates a set of process instances $p_j,...,p_l$, they each inherit the current time from p_i, that is, $\forall x:\{j...l\} \cdot c_x := c_i$.

Because the time readings must be partially ordered, a single integer or real value is inadequate as a data structure.

Thus, when process R spawns processes S and T, they both learn that the counters for P and R have value 1. They also create pairs for themselves the first time they each perform an event (K and I).

• *Rule E: Process termination.* Whenever a set of process instances $p_j,...,p_l$ terminates, the parent process instance p_i merges all the children's logical clocks by maximizing the counter values, that is, $c_i := max(c_i,c_j,...,c_l)$.

In this way process R learns of all activity known to its offspring S and T when they terminate (including updated knowledge of process P). The main program P (which cannot terminate until all processes it has created have terminated) similarly learns from the demise of Q and R.

• *Rule F: Synchronous events.* During a synchronizing event, all process instances involved $(p_j,...,p_l)$ maximize their local clocks using the counters from every other participating process instance, that is, $\forall x:\{j...l\} \cdot c_x := max(c_j,...,c_l)$. A computation applies this rule only after any increments required by rule B.

Thus, during the synchronous message-passing action represented by events F and I, process S learns of the time in process P and vice versa: Logical time information is exchanged in both directions (hence the double-headed arrow in Figure 1). This happens because the synchronization resulting from unbuffered message-passing is symmetric: Both sender and receiver block. Figure 1 shows only a biparty interaction, but rule F allows for any number of synchronizing processes.

Asynchronous communication is asymmetric (only receivers block) and hence requires two rules:

• *Rule G: Sending.* Whenever a process instance p_i sends a message, that message carries the current value of c_i. A process applies this rule only after any increments required by rule B.

• *Rule H: Receiving.* Upon receiving a message, the receiving process instance p_j maximizes its counter values using those received in the piggybacked time stamp $c_{received}$, that is, $c_j := max(c_j, c_{received})$.

Thus event C allows process Q to learn of those events known to process P when event G was performed. Although Figure 1 shows only a biparty communication, these rules also apply to broadcast messages.

The logical clock information is always piggybacked onto *existing* communication pathways and thus must reflect the structure of the computation.

• *Rule I: Time-stamping.* The time stamp t_{e_i} associated with the execution of event e_i by process instance p_i, is the value of c_i immediately following the application of rules B through H.

Finally, with these rules in place, we can determine the "happened before" relations using the comparison property of such time stamps:

• *Comparison property.* Given two time stamps t_{e_i} and t_{f_j}, event e_i happened before event f_j if and only if t_{f_j} has knowledge of process p_i as recent as the execution of e_i, but not vice versa, that is,

$$e_i \rightarrow f_j \Leftrightarrow (t_{e_i}(i) \leq t_{f_j}(i)) \wedge (t_{e_i}(j) < t_{f_j}(j))$$

We need to compare only two pairs from the sets to establish whether there is a causal relationship between the two events. The first comparison is true if and only if e_i can causally affect f_j. The second comparison precludes reflexivity because it would be nonsensical to say that an event happened before itself.[1] (Because synchronously communicating processes may independently time-stamp their part of a shared event, as do F and I in Figure 1, it may not always be easy to directly test $e_i \neq f_j$.)

Elsewhere I have formally shown that

these rules are sufficient to implement the "happened before" relation.[2] The rules are robust enough for asynchronous message "overtaking" (that is, non-FIFO queuing of messages destined for a particular process instance). The rules also preserve the equivalence between asynchronous message-passing and synchronous message-passing with an intervening buffer process, and between synchronous message-passing and a single event shared among the communicating processes.

As Figure 2 shows, substituting the appropriate counter values from the time stamps into the formula from the comparison property establishes the relationships defined by the computation in Figure 1. The final category in Figure 2 shows the principal advantage of partially ordered logical clocks over previous time-stamping methods.[1] Where no causal relationship exists between events, no arbitrary ordering is imposed on them. Thus we can tell, for instance, that events K and J could occur in either order, or at exactly the same time. Even though Figure 1 suggests that K occurred before J in real time (taking a line drawn horizontally through the diagram to represent an instant in global real time), the logical behavior of the computation does not enforce this temporal ordering. A slightly different interleaving of the same computation may result in J occurring before K. (Think of the dots in Figure 1 as beads free to slide up and down the time lines, as long as they do not violate causality by, for example, causing communication arrows to point backward.)

Tests for $e \rightarrow f$ where

- e and f are the same event:
  $(2 \leq 2) \wedge (2 \not< 2) \Rightarrow \neg(C \rightarrow C)$
- e and f are different events in the same process:
  $(1 \leq 3) \wedge (1 < 3) \Rightarrow B \rightarrow D$
  $(2 \not\leq 1) \wedge (2 \not< 1) \Rightarrow \neg(C \rightarrow B)$
- e and f are in different processes with an intervening communication action:
  $(2 \leq 4) \wedge (0 < 3) \Rightarrow E \rightarrow D$
  $(2 \not\leq 0) \wedge (3 \not< 2) \Rightarrow \neg(J \rightarrow E)$
- e is in a subprocess of the process containing f or vice versa:
  $(1 \leq 1) \wedge (0 < 2) \Rightarrow H \rightarrow J$
  $(5 \not\leq 3) \wedge (2 \not< 1) \Rightarrow \neg(M \rightarrow I)$
- e and f are different parts of a synchronous communication event:
  $(3 \leq 3) \wedge (1 \not< 1) \Rightarrow \neg(F \rightarrow I)$
  $(1 \leq 1) \wedge (3 \not< 3) \Rightarrow \neg(I \rightarrow F)$
- e and f are "potentially concurrent"; that is, there is no causal relationship between them:
  $(2 \not\leq 0) \wedge (1 < 2) \Rightarrow \neg(C \rightarrow J)$
  $(2 \not\leq 1) \wedge (0 < 2) \Rightarrow \neg(J \rightarrow C)$
  $(1 \not\leq 0) \wedge (0 < 2) \Rightarrow \neg(K \rightarrow J)$
  $(2 \not\leq 0) \wedge (0 < 1) \Rightarrow \neg(J \rightarrow K)$

Figure 2. Some "happened before" relationships defined between two arbitrary events e and f by the time stamps in Figure 1.

Optimizations

In their full generality, as described in the previous section, partially ordered logical clocks may be impractically expensive for long-lived computations. For instance, the rules placed no upper bound on the size of the set of pairs, and their number was limited only by the number of process instances created at runtime. Nevertheless, several optimizations are possible, depending on the application environment in which the clocks will be used.

Static number of processes. Where there is no process-nesting, and the system knows the number of processes to be created at compile time, the "set of pairs" data structure is unnecessary. Instead, each process can use an array of counters, with one element reserved per process.[3]

This frequently used optimization is known as "vector time"[3] because each clock reading is a vector (array) of counter values. It has the obvious advantage of placing an upper bound on the storage requirements for the auxiliary clock variables. In a formal proof, Charron-Bost demonstrated that a vector of length n is minimal for n static processes.[4]

The vector-time optimization can be applied to languages with nested concurrency if they do not allow recursive process definitions.[2] This restriction means that only one instance of each static process definition can execute at any given time. The system can determine from the source code the maximum number of runtime process instances simultaneously executing and reserve only one vector element for each. Every time a particular process definition is instantiated, it uses the same element in the logical clock vector, because no other copies of itself are currently running.

Comparisons known a priori. So far we have assumed that the computation maintains counters for every process instance. However, if we know in advance the processes that contain the events we wish to study, then we need to keep counter values only for those processes.[5] Nevertheless, other "uninteresting" processes must still maintain an auxiliary clock variable and transfer information at communication events. Otherwise, the partial ordering may fail to correctly reflect transitive interprocess dependences. All processes and all synchronizing actions in a computation must actively participate.[1]

Only new values piggybacked. When the number of processes in the vector-time model is large, the transmission of the clock arrays during message-passing represents a significant overhead. In such cases each process can main-

tain, via two further auxiliary arrays, the value of the "local" counter when a vector was last sent to each other process, and when each counter for other processes was last updated. Using this information, the process can piggyback on an outgoing message only those counter values that have been modified since the last communication with the target process, assuming message overtaking is precluded.[6]

Implementations and applications

Partially ordered logical clocks have been used in a number of practical and theoretical areas.

Languages. Inmos's Occam has only one interprocess communication mechanism, synchronous message-passing, and nested concurrency without recursion. Thus, programmers can easily add partially ordered clocks. I have experimentally introduced "logical timers" in a way consistent with Occam's existing real-time timer.[2]

So far we have assumed that message-passing is the only interprocess communication medium. Languages that allow interprocess synchronization in other ways — for example, through access to shared memory, monitors, and semaphores — must also incorporate rules for such synchronization.

Bryan has defined partially ordered time for Ada.[7] In Ada, there are several unusual ways that tasks (processes) define causal relationships between events. The Ada rendezvous causes two tasks to synchronize while the "accept" code is executed, after which independent execution continues. One task can unconditionally "abort" another. Unhandled exceptions may propagate to another task. When shared variables are used, the Ada standard guarantees synchronization only at certain points in the computation. Bryan has formally defined the causal relationships for all of these activities.

Debugging distributed systems. Programmers trying to debug distributed computing systems are faced with a frustrating inability to see what is happening in the network of processes.[2] To detect the occurrence of events in geographically distant processes, an observer must receive a "notification" message. Because of unpredictable propagation delays, the arrival times of these notifications may bear no resemblance to the order in which the events originally occurred. Time-stamping the notifications at their source with the current real time is also unhelpful, even when the local real-time clocks are closely synchronized, because the real-time ordering of events may be affected by CPU loads unrelated to the computation of interest. The perceived ordering of events based on these time stamps may be merely an artifact of relative processor speeds, with no significance for the computation itself, and may be different each time the same computation is performed.

In the past, a popular approach to this problem was to time-stamp events using so-called Lamport clocks (totally ordered logical clocks).[1] Unfortunately, these time stamps impose on unrelated concurrent events an arbitrary ordering that the observer cannot distinguish from genuine causal relationships.

Time-stamping the event notifications with partially ordered time readings resolves all the debugging problems. The observer receives an accurate representation of the event orderings, can see all causal relationships, and can derive all possible totally ordered interleavings. Most importantly, the technique greatly reduces the number of tests required. It is never necessary to perform the same computation more than once to see whether different event orderings (interleavings) are possible.[2]

Partially ordered logical clocks have been used experimentally for the detection of global conditions in a homogeneous network of processors,[8] for two prototype implementations of a monitor for Ada programs,[5] and for a prototype temporal assertion checker for Occam.[2]

Definition of global states. The "happened before" relation provides for straightforward definitions of normally subtle concepts. For instance, a "cut" of a distributed computation partitions the events performed into two sets: "past" and "future." In a consistent cut, the set of past events C does not violate causality; for example, it does not contain the reception of a message without its transmission. This concept has a very simple definition in terms of partially ordered time.[3] Assume that E represents the set of all events performed during a computation using only asynchronous message-passing. Then a consistent cut C is a finite subset $C \subseteq E$ such that

$$\forall e: E; c: C \cdot (e \rightarrow c) \Rightarrow (e \in C)$$

In other words, if any event e "happened before" an event c in the cut set C, then e must also be in the cut set.

This is an important concept in the theory of distributed error recovery and rollback. Consider a distributed system consisting of a static number of nonnested processes, each of which periodically stores a "snapshot" of its local state (including the contents of message queues). If a set of snapshot events $S \subseteq C$, one from each process, forms the leading edge of a consistent cut C, that is,

$$\forall s: S; c: C \cdot \neg (s \rightarrow c)$$

then these local states form a valid global state from which an erroneous computation may be restarted.

Partially ordered time has also been used in the analysis of other global state problems, for instance, characterization of distributed deadlocks.[9]

Concurrency measures. A concurrency measure is a software metric that objectively assesses how concurrent a computation is. It measures the structure of the computation graph, rather than elapsed execution time. Partially ordered logical clocks have proved important in the definition and proposed implementations of such measures.

One of the simplest such measures, known as ω, counts the number of concurrent pairs of events that occurred during the computation and divides this by the total number of pairs of events between processes.[10] For a given computation C, consisting of two or more nonnested processes, the measure is

$$\omega(C) = \frac{\left| \left\{ (e_i, f_j) : e_i \text{ co } f_j \right\} \right|}{\left| \left\{ (e_i, f_j) : i \neq j \right\} \right|}$$

where the relation "co" is true between two distinct events if and only if they cannot causally affect one another, that is,

$$e_i \text{ co } f_j \Leftrightarrow \neg (e_i \rightarrow f_j) \wedge \neg (f_j \rightarrow e_i)$$

Charron-Bost[10] has defined a more discerning measure, m, using consistent cuts:

$$m(C) = \frac{\mu - \mu^s}{\mu^c - \mu^s}$$

The value μ represents the number of consistent cuts that occurred during the computation, μ^s is the number of consistent cuts that would be possible if the computation consisted of only one process, and μ^c is the number of consistent cuts possible if causal relationships due to interprocess communication are ignored.

Because both ω and m are defined in terms of the "happened before" relation, they can both be implemented by time-stamping all events with partially ordered time readings for postmortem analysis. Elsewhere I have investigated measures that allow for process-nesting or can be evaluated efficiently at runtime.[11]

Enforcement of causal ordering. The "causal ordering" abstraction, which prevents asynchronous message overtaking, has applications in several areas.[12] For management of replicated data in distributed databases, it can be used with a "write-enabling" token model to ensure that updates are applied in the same order at all sites. When monitoring activity in distributed systems, it ensures that all observers receive notification of events in the same order. Also, in the allocation of shared resources, causal ordering guarantees that servers honor requests in the order that they were made, rather than received.

The rule required for enforcement of causal ordering is easily defined in terms of "happened before."[12] For two send events e_i and f_j, where both messages are received by process p_k as events g_k and h_k, respectively, we need to guarantee that

$$(e_i \rightarrow f_j) \Rightarrow (g_k \rightarrow h_k)$$

A possible implementation of this rule using vector time (for computations without process-nesting) involves piggybacking control information on each outgoing message m, so the destination process knows whether there are other messages in transit that it must receive before it can accept m.[12] The piggybacked information consists of a bounded number of destination-site/vector-time pairs representing messages known to be in transit to the destination process. A destination process must not accept a message until all the message's time stamps "happened before" the current local time.

Partially ordered logical clocks are a fundamental new approach to the analysis and control of computations performed by distributed computing systems. They accurately reflect causality and are unperturbed by the random influences of system load, relative processor speeds, and different system configurations. In testing and debugging, they greatly reduce the number of tests required by simultaneously presenting any observer with all possible interleavings of events. Both their theory and practical application are now well established, but we will see further progress in both areas in the near future. ■

Acknowledgments

Thanks to Friedemann Mattern, Michel Raynal, Bernadette Charron-Bost, Doug Bryan, Dieter Haban, Sigurd Meldal, and Mukesh Singhal for keeping me abreast of their work on partially ordered clocks. Special thanks to Doug Bryan, Andrew Lister, and the anonymous referees for their numerous helpful comments on drafts of this article.

This work was supported by an Australian postdoctoral research fellowship and an Australian Telecommunications and Electronics Research Board project grant.

References

1. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, Vol. 21, No. 7, July 1978, pp. 558-565.
2. C.J. Fidge, *Dynamic Analysis of Event Orderings in Message-Passing Systems*, doctoral dissertation, Australian Nat'l Univ., 1989.
3. F. Mattern, "Virtual Time and Global States of Distributed Systems," in *Parallel and Distributed Algorithms*, M. Cosnard and P. Quinton, eds., North-Holland, Amsterdam, 1989, pp. 215-226.
4. B. Charron-Bost, "Concerning the Size of Clocks," Tech. Report 569, Laboratory of Research in Information Science, University of Paris-South, Paris, 1990.
5. S. Meldal, "Supporting Architecture Mappings in Concurrent Systems Design," *Proc. Fifth Australian Software Eng. Conf.*, IREE, Sydney, 1990, pp. 207-212.
6. M. Singhal and A. Kshemkalyani, "An Efficient Implementation of Vector Clocks," tech. report, Dept. of Computer and Information Science, Ohio State Univ., Columbus, Ohio, 1990.
7. D. Bryan, "An Algebraic Specification of the Partial Orders Generated by Concurrent Ada Computations," *Proc. TriAda*, ACM Press, New York, 1989, pp. 225-241.
8. D. Haban and W. Weigel, "Global Events and Global Breakpoints in Distributed Systems," *Proc. 21st Hawaii Int'l Conf. System Sciences, Vol. II*, IEEE Computer Society Press, Order No. 842 (microfiche only), 1989, pp. 166-175.
9. A.D. Kshemkalyani and M. Singhal, "Characterization of Distributed Deadlocks," Tech. Report OSU-CISRC-6/90-TR15, Computer and Information Science Research Center, Ohio State Univ., Columbus, Ohio, 1990.
10. B. Charron-Bost, "Combinatorics and Geometry of Consistent Cuts: Application to Concurrency Theory," in *Distributed Algorithms*, J.-C. Bermond and M. Raynal, eds., *Lecture Notes in Computer Science*, Vol. 392, Springer-Verlag, Berlin, 1989.
11. C.J. Fidge, "A Simple Run-Time Concurrency Measure," in *The Transputer in Australasia (ATOUG-3)*, T. Bossomaier, T. Hintz, and J. Hulskamp, eds., IOS Press, Amsterdam, 1990, pp. 92-101.
12. M. Raynal and A. Schiper, "The Causal Ordering Abstraction and a Simple Way to Implement It," Tech. Report 1132, INRIA, Paris, 1989.

Colin Fidge is a postdoctoral fellow with the Key Centre for Software Technology, University of Queensland, where he researches techniques for the specification and development of distributed real-time systems. He has also worked in the Software Engineering Research Section of the Telecom Australia Research Laboratories.

Fidge completed his PhD in computer science at the Australian National University in 1989.