Efficient Vector Time with Dynamic Process Creation and Termination

Golden G. Richard III
Dept. of Computer Science
University of New Orleans
New Orleans, LA 70148
Email: *golden@cs.uno.edu*

Abbreviated Title : “Efficient Vector Time with Dynamic Processes”

Contact information for author:

Prof. Golden G. Richard III
Dept. of Computer Science
University of New Orleans
New Orleans, LA 70148

(504) 280-6594 (secretary)
(504) 280-6045 (direct)
(504) 280-7228 (FAX)

Email: golden@cs.uno.edu

Abstract

Many distributed algorithms require knowledge of the causal relationships between events. Examples include optimistic recovery protocols, distributed debugging systems, and causal distributed shared memory. Determining causal relationships can be difficult, however, because there is no global clock and local clocks cannot be perfectly synchronized. Vector time is a useful abstraction for capturing the causal relationships between events and, unlike Lamport’s logical clocks, allows identification of concurrent events. Some drawbacks of vector time include transmission and logging overhead, since the size of a vector clock is linear in the number of processes. This paper presents a technique to reduce these overheads for applications that dynamically create and destroy processes and log event information with attached vector timestamps. The reduction in logging overhead comes at the expense of a more complicated timestamp comparison protocol and more sophisticated data structures for maintaining vector time. Distributed process recovery mechanisms and debugging systems that require “on-the-fly” causality information can benefit directly from the proposed technique.

Typesetting Information

There are no non-standard characters in the document, aside from standard mathematical symbols such as $\in$ (set membership), $\forall$ ("for all"), $\exists$ ("there exists"), etc.

1. Introduction

We model a distributed system as a set of processes that communicate with each other through logical unidirectional, FIFO, point-to-point channels. A physical processor may support more than one process. For descriptive purposes, we assume that the system is logically completely connected.

An important characteristic of distributed systems is that there is no global clock. Consequently, ordering the events (which consist of internal events, message sends, and message receives) in a distributed system can be challenging. Lamport [7] introduced an efficient mechanism called *logical clocks* for totally ordering the events in a distributed system, but the mechanism is not sufficiently powerful to allow concurrent events to be identified. Mattern [8] and Fidge [3] independently developed *vector clocks*, which precisely capture the causal ordering between distributed events. Under the Mattern and Fidge schemes, each process maintains a vector $TS[1..N]$ of integral logical timestamps, indexed by process identifiers (beginning at 1), where N is the number of processes in the system. Element $TS[i]$ is the logical clock for process i and for each $i \neq j$, $TS[j]$ is the largest logical clock value for process j upon which process i causally depends. The individual logical clocks are essentially event counts. Vector time is maintained by a process i as follows:

- ✓ Initially, the vector clock is initialized to all zeros.
- ✓ Events at process i cause $TS[i]$ to be incremented by 1.
- ✓ When process i sends a message m, the current value of TS is attached to the message (this is denoted by $m.TS$). The attached vector timestamp provides complete transitive dependency information for the message send event.
- ✓ When a message m is received, then $\forall j \neq i$, $TS[j]$ is set to $\max(TS[j], m.TS[j])$. This allows process i to track its transitive dependencies on all other processes in the system.

Vector clocks are compared using the vector operation “<”, which performs a component-wise comparison. For vector timestamps $T1$, $T2$:

- ✓ $T1 \leq T2$ iff $\forall k$, $T1[k] \leq T2[k]$.
- ✓ $T1 < T2$ iff $T1 \leq T2$ and $\exists k$ with $T1[k] < T2[k]$.

An event *e2* causally depends on an event *e1* iff $e1.TS < e2.TS$. Events *e1* and *e2* are concurrent iff neither $e1.TS < e2.TS$ nor $e2.TS < e1.TS$.

Vector time is useful in many distributed applications that require causal information, including distributed debugging [1, 5] and some recently proposed optimistic recovery techniques [10, 11]. Mattern [8] discusses the usefulness of vector time extensively. However, existing vector time schemes have three drawbacks:

1. Attaching vector timestamps results in larger messages, increasing communication overhead,
2. In applications that log messages or event information to a slow, stable medium, the attached vector timestamps can substantially increase logging overhead, and
3. Drawbacks (2) and (3) are aggravated in the presence of dynamic process behavior (process creation and termination).

All the drawbacks arise essentially because the size of a vector clock is linear in the number of processes. In fact, Charron-Bost [2] has proven that vector clocks that correctly characterize causal order must be of length N, since a scenario can always be created where some causal relationships are lost if shorter vectors are used. Torres-Rojas [17] has investigated fixed-length vector clocks, called *plausible clocks*, which violate Charron-Bost's "length N" rule. These clocks are cheaper to implement than full vector clocks, but in some cases fail to reflect accurate causal relationships. They are useful in the implementation of causal distributed shared memory protocols and other applications where missing causal information results only in a performance tradeoff, but plausible clocks are not replacements for full vector clocks.

The dependency-tracking scheme used by Fowler and Zwaenepoel [5] attaches only a single integer to each message, but construction of a vector timestamp for an event requires a recursive trace of the direct dependencies. This can be very time consuming and is appropriate for "off-line" applications. Singhal and Kshemkalyani [16] addressed the first drawback for static process systems with a technique that attaches only differential vector clock information to messages. Processes reduce communication overhead by transmitting only portions of their vector clock that have changed since the last message transmission to a particular process.

Existing techniques simply increase the vector clock size as new processes are encountered, either through direct or indirect communication [3]. Further, the techniques for handling process termination under vector time [4] do not reduce the size of vector clocks at all. For applications that log messages with attached vector timestamps, this causes logging overhead to grow in the presence of dynamic process behavior. However, specifically addressing dynamic process behavior can limit the unbounded increase in vector clock size as processes are created and terminated. Our work provides a unified technique for addressing all the drawbacks of vector time together:

- ✓ Communication overhead is reduced by integrating Singhal and Kshemkalyani's technique, and
- ✓ Vector clocks are reduced in size when processes terminate, so logging overhead is reduced.

The proposed technique is aimed at reducing logging overhead for applications that dynamically create and terminate processes while logging messages or event information with complete, attached vector timestamps. It provides no benefit for static process systems over the technique of Singhal and Kshemkalyani. Vector time-based distributed process recovery techniques [10, 11] will especially benefit from the proposed technique, since they log large numbers of messages with attached vector timestamps and require access to the timestamps "on-the-fly" during recovery.

The rest of the paper is organized as follows: Section 2 presents our technique. A correctness argument is included in Section 3. Section 4 analyzes the technique and discusses optimizations and garbage collection. Finally, conclusions are presented in Section 5.

2. Reducing Logging Overhead for Vector Time with Dynamic Processes

Our goal is to reduce logging overhead for applications that exhibit dynamic process behavior and log messages or events tagged with vector timestamps. This is accomplished by removing information about terminated processes from vector clocks as process terminations are observed. Of course, the deletion of process information from data structures that support the vector time abstraction must not affect the determination of causal relationships between events. To illustrate this concern, consider the example in Figure 1. Process 1 has sent message A to process 2 and message B to process 3. The vector clocks at processes 2 and 3 will show the causal dependency on process 1 after the reception of A and B. Now imagine that when processes 2 and 3 learn of the termination of process 1, the entries for process 1 are simply deleted from the vector clocks at processes 2 and 3. Then process 3 sends message C to process 2. The causal relationship between the send events for messages A and C is permanently obscured (these events appear to be concurrent from examination of their attached vector timestamps). Thus, a naïve approach is not the answer. Our technique preserves causal information while deleting entries corresponded to terminated processes from vector clocks by introducing some additional data structures to track terminated processes. We describe a method for augmenting vector timestamps before comparison to temporarily reintroduce information about terminated processes.

2.1. Data Structures

For clarity, we use vector notation for vector clocks and associated data structures, even though an implementation of the protocol will probably use sparse { (*clock*, *PID*), (*clock*, *PID*), … } representations. We introduce the convention *TS[j]=nil* to denote that there is no element with index *j* in the sparse vector *TS*.

Each process *i* maintains the following data structures:

- ✓ *PIDi* - the unique process identifier for process *i*. An abstract identifier like "*i*" and the identifier *PIDi* will be interchangeably.
- ✓ *TSi* - the vector clock for the process, containing one Lamport-style clock value for each live process in the system upon which *i* is causally dependent. The value ∞ represents the termination event for a process.
- ✓ *LUi* (Last Update) - a vector with *LUi[j]* equal to the value of *TSi[i]* when *i* last updated *TSi[j]*.
- ✓ *LSi* (Last Sent) - a vector with *LSi[j]* equal to the value of *TSi[i]* when *i* last sent a message to process *j*.
- ✓ *LRi* (Last Received) - a vector of vector clock structures (like *TSi*, above), one for each live process (except *i* itself) from which *i* has received a message. *LRi* reflects process *i*'s local view of the vector clocks of other processes. This structure allows the complete vector timestamp for the send event of an incoming message to be reconstructed.
- ✓ *TLi* (Process Termination List) - a vector with *TLi[j]* equal to the value of *TSi[i]* when process *i* witnessed the termination of *j*.
- ✓ *STLi* (System-wide Termination Lists) - a set of *TL* structures (like *TLi*, above), one for each process other than process *i* (including terminated processes). Each is current as of the last time the corresponding process communicated with process *i*. *STLi[j][k]* is the value of *TSj[j]* when *j* witnessed termination of process *k*. Entries in *STLi* reflect *i*'s best knowledge of the values of the *TL* vectors at other processes.

For each event (internal, process creation, message transmission, or message reception), process *i* updates its vector clock by incrementing *TSi[i]* and setting *LUi[i] = TSi[i]*. The latter ensures that the updated *TSi[i]* will be sent to other processes when *i* communicates with them.

Specific actions required for message transmission, message reception, process creation, and process termination are described below.

Message Transmission

When a process *i* sends a message *m* to a process *j*, two sparse vectors are attached to the message. The attached vectors *m.CLOCKS* and *m.TERMINATED* are determined as follows:

- ✓ *m.CLOCKS* consists of all elements *TSi[k]* such that *LSi[j] < LUi[k]*. These are elements of *TSi* which have been modified since process *i's* last message transmission to process *j*.
- ✓ *m.TERMINATED* consists of all elements *TLi[k]* such that *TLi[k] > LSi[j]*. These are process terminations witnessed by process *i* since process *i*'s last message transmission to process *j*.

Before the message *m* is transmitted, but after *m.CLOCKS* and *m.TERMINATED* are constructed, *LSi[j]* is set to *TSi[i]* to ensure that the next transmission to *j* will carry a minimal amount of overhead.

Message Reception

Messages fall into two categories, normal messages and *termination notification* messages. We consider only normal messages here. Termination notification messages are considered under Process Termination, below. When a process *i* receives a message *m* from a process *j*, the attached vectors *m.CLOCKS* and *m.TERMINATED* are examined. These contain differential vector clock information and differential process termination information from process *j*.

For each element *m.CLOCKS[k]*:

- ✓ If *TLi[k]=nil*, then *TSi[k]* is set to max(*TSi[k]*, *m.CLOCKS[k]*) and *LUi[k]* is set to *TSi[i]* if *TSi[k]* is actually updated. The exclusion of processes whose *PID* is present in *TLi* prevents information about terminated processes from being reintroduced after their termination has been witnessed.
- ✓ If *STLi[j][k]=nil*, then *LRi[j][k]* is set to max(*LRi[j][k]*, *m.CLOCKS[k]*). The exclusion is justified as above.

For each element *m.TERMINATED[k]*:

- ✓ Process i sets $STLi[j][k]$ to $m.TERMINATED[k]$.
- ✓ If $TLi[k]=nil$, then $TLi[k]$ is set to $TSi[i]$ and elements $TSi[k]$, $LUi[k]$, and $LSi[k]$ are removed. This causes process i to (indirectly) witness termination of processes whose termination j has witnessed.
- ✓ Process i removes the element $LRi[j][k]$.

At this point, the *CLOCKS* and *TERMINATED* attachments can be discarded. $LRi[j]$ is the vector timestamp for the send event of m and can be attached to the message data in m and logged to whatever application-specific media is appropriate. The vector $LRi[k]$ cannot be deleted based on i indirectly witnessing the termination of k, because messages addressed to i from k may still be undelivered. Instead, $LRi[k]$ will be deleted when i itself receives a termination notification from k. This is discussed further under Process Termination, below.

Process Creation

When a process i creates a process j, the new process is casually dependent on the same events as process i. This is reflected in the data structures at process j as follows:

- ✓ Process j inherits the vector timestamp of its parent, supplemented with an additional element for j itself, which is initialized to 0.
- ✓ LRj is initialized to contain a single element by setting $LRj[i]$ to TSi. This allows complete vector timestamps for future messages from process i to be assembled.
- ✓ TLi is initialized so that process j witnesses all terminations witnessed by process i, but at time 0. That is, for every element $TLi[k]\neq nil$, the element $TLj[k]$ is set to 0. In addition, $STLj[i]$ is set to TLi.

Process Termination

Before a process j exits, it transmits a final message to live processes with which it has communicated. A process i receiving such a message witnesses the termination of j by deleting the elements $TSi[j]$, $LUi[j]$, and $LSi[j]$ and setting $TLi[j]$ to $TSi[i]$ (unless $TLi[j]\neq nil$; this can occur because process termination can be observed indirectly via communication with processes that have already witnessed terminations). The vector $LRi[j]$ can be deleted after the value is attached and logged with the termination message (as appropriate). Due to the FIFO requirement for communication channels, no additional (delayed) messages addressed to *i* from process *j* are possible. Note that termination messages carry the *CLOCKS* and *TERMINATED* sets; essentially, a termination message is a null message (or even a final application message) *m* with an element in *m.TERMINATED* corresponding to the sending process *j*. A side effect of the explicit termination messages is that processes become causally dependent upon witnessed termination events.

2.2. Comparing Vector Timestamps

Under the proposed technique, vector timestamps do not necessarily contain elements for every process. For a particular event *e* executed at process *j*, the vector timestamp *e.TS* may be missing elements for terminated processes and for processes with which *j* has established no causal relationships. Before the causal relationship between two events *e1* and *e2* can be determined, the vector timestamps *e1.TS* and *e2.TS* must be augmented to reintroduce entries for terminated processes and to normalize the process sets that are represented.

Let *e1.PID* and *e2.PID* be the processes that executed the events *e1* and *e2*. Suppose process *i* wishes to determine the relationship between *e1* and *e2*. Process termination lists for *e1.PID* and *e2.PID* are present in *STLi[e1.PID]* and *STLi[e2.PID]*. The following augmentations are necessary before *e1.TS* and *e2.TS* can be compared:

- ✓ For each S*TLi[e1.PID][k]* such that *e1.TS[k]=nil* and *STLi[e1.PID][k]* ≤ *e1.TS[e1.PID]*, the element *e1.TS[k]* is set to ∞. A similar augmentation is performed on *e2.TS*, using *STLi[e2.PID]*. This augmentation reintroduces elements for processes whose termination was witnessed by either *e1.PID* or *e2.PID* before the execution of *e1* or *e2*, respectively. It is not necessary to introduce an ∞ value for a particular process *k* into *both e1.TS* and *e2.TS.*

- ✓ For each element *e2.TS[k]≠nil* with *e1.TS[k]=nil*, the element *e1.TS[k]* is set to 0. A similar augmentation is applied to *e2.TS*. This augmentation introduces "place holder” elements to normalize the process sets with which *T1.PID* and *T2.PID* established causal relationships.

After these steps, timestamps *e1.TS* and *e2.TS* are of the same length and contain elements for the same process sets. Then event *e1* causally precedes event *e2* iff *e1.TS* < *e2.TS.* The events are concurrent iff neither *e1.TS* < *e2.TS* nor *e2.TS* < *e1.TS.*

2.3. Example

Figure 2 and Table 1 illustrate the proposed technique. Processes 1 and 2 exist from the beginning of the computation, while processes 3, 4, and 5 are dynamically created. Processes 3 and 4 terminate at the points indicated; termination messages are dotted in the figure. Table 1 shows the *CLOCKS* and *TERMINATED* attachments for each message transmitted, as well as the vector timestamp for the send event, minus information about terminated processes. The latter is included to assess logging overhead. Recall that information about terminated processes need not be logged; complete timestamps can be reconstructed on demand with local information. At the height of the computation, vector timestamp size reaches length 5: the timestamp for message F to be logged at process 1 is [$1\rightarrow1$ $2\rightarrow5$ $3\rightarrow3$ $4\rightarrow3$ $5\rightarrow2$]. After propagation of the termination information for processes 3 and 4, a reduction in total logging overhead over a non-dynamic scheme becomes apparent. The timestamps to be logged for messages H-L are of length 3, with causal information related to the terminated processes 3 and 4 maintained outside the log. Clearly, additional communication between processes 1, 2, and 5 will result in additional savings. Detailed tables that completely define the various data structures for each event in this example appear in [14]. These tables are suitable for "turning the crank" to see the exact effect of each event.

3. Correctness Argument

We now argue informally that the proposed technique correctly captures the causal relationships between events. The central point of the argument is that the proposed technique can recreate correct vector timestamps for events despite the transfer of differential vector timestamp information and the removal of information corresponding to terminated processes from vector clocks.

Property 1. *When a message m is received from a process j, the complete vector clock value for j corresponding to m's send event, disregarding terminated processes, is available at the receiving process i.*

This follows directly from the technique for updating *LRi*. Processes transfer differential vector clock information when they communicate, based on the last sent and last update variables, *LSi* and *LUi*. When a process updates its vector clock, either with a simple increment before a message send or internal event, or when a message is received, the event count when the modification is made is noted in *LUi*. When a process *j* sends a message to a process *i*, a comparison of the event count when *j* last communicated with *i* and the event counts when portions of *j*'s vector clock were updated yields the portions of *j*'s vector clock which must be transfered to *i*. This allows complete vector clock values for a message sender to be built at the receiver.

<u>Property 2.</u> *When a message m is received by a process i from a process j, the list of processes whose termination j observed before the send event for m is available at process i. Further, the event counts when j observed the terminations are available.*

The observation of process termination is transitive information, piggybacked on messages in a differential fashion. The technique used to maintain *STLi* ensures that the process identifiers whose elements have been removed from the complete vector timestamp for *m* are available at the receiver *i*. Determination of differential process termination information to be transmitted when a process sends a message is similar to determination of differential vector timestamp information, discussed above.

<u>Property 3.</u> *Augmentation of a vector timestamp TS in the proposed technique correctly introduces entries for terminated processes.*

When the vector timestamp *e.TS* for an event *e* is augmented, the termination list for the process *e.j* is examined. For all processes *k* which *j* witnessed termination for before the event *e.TS[j]*, an entry for process *k* is potentially missing. For these processes *k*, *e.TS[k]* is set to ∞, which corresponds to the last event in process *k*, namely the termination notification messages sent immediately before its termination.

<u>Theorem.</u> *For events e1 and e2, event e1 causally precedes e2 iff e1.TS < e2.TS, where e1.TS and e2.TS are the augmented vector timestamps associated with events e1 and e2.*

This result follows from the fact that the standard vector time protocol [3, 8] correctly captures the causal relationships between events in a distributed system and Properties 1-3.

4. Discussion

4.1. Performance

The proposed technique trades off a larger memory footprint, some additional messages, and a more complicated vector timestamp comparison protocol to reduce logging overhead to slow media (e.g., disks). This tradeoff is appropriate because processor speeds, memory costs, and network bandwidth are becoming less burdensome at a faster rate than speedup of mechanical logging media.

Regarding the savings in logging overhead, consider the following parameters:

p: Number of bits per *PID*

c: Number of bits per clock entry

L: Number of live processes

T: Number of terminated processes

In an implementation of the proposed protocol, sets of (*PID*, *clock*) pairs will be used to represent vector timestamps. The savings (in bits) per message logged is potentially $O(T(p + c))$, since our technique logs only vector timestamp information associated with live processes. A naïve logging technique would log a maximum of $(L+T)(p + c)$ bits per message. The savings comes at the expense of $O((L+T)^2 * (p + c))$ bits of memory per process for data structures, disregarding the potential savings possible with garbage collection, discussed below. Note that the Singhal-Kshemkalyani [16] technique requires $O((L+T)^2 * (p + c))$ bits of memory per process just to support differential vector time if complete vector timestamps must be reconstructed at the receiver. The proposed technique will be most useful for applications where logging of messages or events with attached vector timestamps is essential. As with the Singhal-Kshemkalyani technique, mostly localized communication patterns between small clusters of processes will yield the greatest benefits. The communication overhead of the proposed technique exceeds that of Singhal-Kshemkalyani only when process termination information is exchanged. The memory requirements of the proposed technique and that of Singhal-Kshemkalyani are of the same order.

4.2. Optimizations and Garbage Collection

Several optimizations are possible. A logically completely connected system was assumed for ease of exposition, but processes that do not develop causal relationships need not track information about each other [4, 15]. In some circumstances, this can substantially reduce the size of the data structures used to maintain the vector time abstraction.

One problem with the proposed technique is that *TLi* and *STLi* can grow without bound as processes terminate and new processes are created. In general, the size of these data structures is offset by reduced logging overhead. Still, in long-lived applications and in applications with very dynamic process behavior, garbage collection may be desirable. The condition for garbage collection for *TLi* and *STLi* is simple to describe, but the circumstances under which it is satisfied are application-specific. At a process i, before an entry *TLi[k]* or *STLi[j][k]* can be deleted, it must be possible to guarantee that all processes have witnessed the termination of process *k* and further, that all vector timestamps to be compared in the future were assembled after the associated processes witnessed termination of *k*.

It is also possible to remove the (potentially large) *STLi* data structures completely. When a process *i* wishes to compare vector timestamps assembled by processes *j* and *k*, it suffices to query these processes to get their current termination lists. This is also useful when a global agent not involved in the distributed computation wishes to compare vector timestamps. Of course, the termination list of a *terminated* process *k* cannot be obtained by querying *k* (the dead never speak). Having terminating processes save their termination list on stable storage or broadcast it before termination are possible solutions if this direction is pursued.

5. Conclusions

The major contribution of this work is reduction of logging overhead for applications with dynamic process behavior that log information about events (commonly application messages) with attached vector timestamps. Some applications that will benefit from the proposed technique include vector time-based process recovery [e.g., 10, 11] and some distributed debugging systems. A substantial savings is possible because logged vector timestamps need not contain entries for terminated processes; instead, only those processes which are live at the time a vector timestamp is assembled need to be included. The technique essentially trades a larger memory footprint and a more complicated vector timestamp comparison method for reduced disk transfer times. We believe this is a valid tradeoff, given decreasing memory prices, increasing network bandwidth, and the great disparity between disk I/O and processing speeds.

Our technique is based on the static technique of Singhal-Kshemkalyani. Communication overhead exceeds Singhal-Kshemkalyani's technique only when process termination information is exchanged, and the memory requirement is comparable. The proposed technique yields no improvement over Singhal-Kshemkalyani's technique in applications that do not dynamically create and destroy processes, but is no worse in these applications. One motivation for developing the proposed technique is efficiently supporting vector time in a distributed process recovery testbed [13] that is currently being built at the University of New Orleans.

Acknowledgments

The author thanks Frank Adelstein of the Xerox Design Research Institute at Cornell for reading early versions of this paper and the anonymous referees for suggestions that substantially improved the presentation.

References

[1] T. Basten, “Breakpoints and Time in Distributed Computations,” *Lecture Notes in Computer Science #857*, pp. 340-354, 1994.

[2] B. Charron-Bost, “Concerning the Size of Logical Clocks in Distributed Systems,” *Information Processing Letters*, vol. 39, no. 1, pp. 11-16, 1991.

[3] C. J. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering," *Australian Computer Science Communications*, vol. 10, no. 1, pp. 56-66, February 1988.

[4] C. J. Fidge, “Logical Time in Distributed Computing Systems,” *IEEE Computer*, vol. 24, no. 8, pp. 28-33, 1991.

[5] J. Fowler, W. Zwaenepoel, "Causal Distributed Breakpoints," *IEEE 10th International Conference on Distributed Computing Systems*, 1990.

[6] G. A. Geist, A. L. Beguelin, J. J. Dongarra, W. Jiang, R. J. Manchek, V. S. Sunderam, *Parallel Virtual Machine-A User’s Guide and Tutorial for Network Parallel Computing*, MIT Press, 1994.

[7] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Communications of the ACM*, vol. 21, pp. 558-565, 1978.

[8] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms*, pp. 215-226, North-Holland, 1989.

[9] F. Mattern, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation," *Journal of Distributed and Parallel Computing*, vol. 18, pp. 423-434, 1993.

[10] S. L. Peterson, P. Kearns, "Rollback Based on Vector Time," *Proc. of the 12th Symp. on Reliable Distributed Systems*, Princeton, NJ, pp. 68-77, October 1993.

[11] G. G. Richard III, M. Singhal, "Complete Process Recovery: Using Vector Time to Handle Multiple Failures in Distributed Systems,” *IEEE Concurrency*, vol. 5, no. 2, pp. 50-59, 1997.

[12] G. G. Richard III, "Techniques for Process Recovery in Message-Passing and Distributed Shared Memory Systems," Ph.D. Dissertation, Dept. of Computer Science, The Ohio State University, 1995.

[13] G. G. Richard III, "A Testbed for Experimental Evaluation of Distributed Process Recovery Mechanisms," Louisiana Education Quality Support Fund Grant # LEQSF(1997-00)-RD-A-28.

[14] G. G. Richard III, "Efficient Vector Time with Dynamic Process Creation and Termination," University of New Orleans Dept. of Computer Science Tech. Report UNOCS-TR-96-03.

[15] R. Schwarz, F. Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *Distributed Computing*, vol. 7, pp. 149-174, 1994.

[16] M. Singhal, A. Kshemkalyani, "An Efficient Implementation of Vector Clocks," *Information Processing Letters*, vol. 43, no. 1, pp. 47-52, August 1992.

[17] F. Torres-Rojas, M. Ahamad, "Plausible Clocks: Constant Size Logical Clocks for Distributed Systems", *Workshop on Distributed Algorithms (WDAG)*, 1996.

Biographical information

The author received the B.S. degree with honors from the University of New Orleans and the M.S. and Ph.D. degrees from The Ohio State University in 1991 and 1995. He has been employed as an Assistant Professor in the Department of Computer Science at the University of New Orleans since August 1994. Dr. Richard's research interests include fault tolerance issues in distributed computing systems, operating systems in general, high-speed networking, and the application of computers to solving problems in criminal justice. He is hopelessly addicted to New Orleans.