Breakpoints and Halting in Distributed Programs

Barton P. Miller
Jong-Deok Choi

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

Abstract

Interactive debugging requires that the programmer be able to halt a program at interesting points in its execution. This paper presents an algorithm for halting a distributed program in a consistent state, and presents a definition of distributed breakpoints with an algorithm for implementing the detection of these breakpoints. The Halting Algorithm extends Chandy and Lamport's algorithm for recording global state and solves the problem of processes that are not fully connected or frequently communicating. The definition of distributed breakpoints is based on those events that can be detected in a distributed system. Events that can be partially ordered are detectable and form the basis for the breakpoint predicates, and from the breakpoint definition comes the description of an algorithm that can be used in a distributed debugger to detect these breakpoints.

Index Items - Distributed Programming, Distributed Debugging, Halting Algorithm, Distributed Breakpoints.

1. Introduction

Interactive debugging requires that the programmer be able to halt a program at interesting points in its execution. Halting consists of the mechanisms to stop the program's execution and the predicates, called breakpoints, that are used to trigger the halting. This paper presents an algorithm for halting a distributed program in a consistent state, and presents a definition of distributed breakpoints with an algorithm for implementing these breakpoints.

Halting a single-process, sequential program is well-understood. There is a single thread of execution that can be stopped without regard for other activities in the system. When a program consists of cooperating processes executing on different machines, halting decisions are affected by unpredictable communication delays between machines. We cannot instantly transmit a command to halt all processes, nor can we guarantee that the halt command will simultaneously reach all processes. In Section 2 of this paper we present an algorithm for consistently halting a distributed program given the inherent

Research supported by the National Science Foundation grant MCS-8105904 and a Digital Equipment Corporation External Research Grant.

communication delays. This algorithm is derived from Chandy and Lamport's algorithm for recording global state [1], and extends this algorithm to work for processes that communicate infrequently or are not fully connected.

Breakpoints in a sequential program have an implied reference to time. When we say ''stop when procedure X is entered *or* when procedure Y is entered'', we mean to stop the program when any of these conditions becomes true. When we say ''stop when procedure X is entered *and* i[j]=7'', we mean to stop the program when, at the same instant, both of these conditions are true.

We have no single, global notion of time in a distributed system [2], so we may not be able to determine whether one condition really occurred before another. This means that we will have to tolerate breakpoints that occur independently on different machines. Likewise, we cannot determine whether events on different machines occurred simultaneously. This means that we must replace the concept of simultaneous events with one that is suitable for a distributed system. In Section 3, we present a definition of predicates for breakpoints in a distributed program. This definition is based on detectable orderings of events. We describe an algorithm from which one can implement a satisfaction detector for these predicates.

Section 4 discusses the application of these ideas to current research in distributed debugging.

2. Consistent Halting

This section describes how to halt all processes belonging to a distributed program so that no critical information is lost when the processes halt. This problem is easy to solve for a single machine because there is only one active process at a given moment. When processes of the same program reside on different machines, they cannot be stopped simultaneously. Therefore, some information may be lost or recorded incorrectly.

Our halting algorithm is derived from Chandy and Lamport's algorithm for recording global states [1]. We first summarize Chandy and Lamport's algorithm and then present an algorithm to halt the distributed computation in such a way that, in spite of the time delay in halting processes, the final halted states of the processes of the computation result in globally consistent states. Although the physical instant of halting each process by our algorithm is different, we show that all the processes halt at the same virtual time instant [2]. For any two halted processes of a computation, the halted state of a process is not affected by the halted state of the other process and, therefore, there can be no *happened-before* [2] relationship between the two halted states. Each process's view of event ordering is preserved by our algorithm.

We show some problems with this basic halting algorithm and then present an extended algorithm that is suitable for a debugger.

2.1. Chandy and Lamport's Algorithm

A distributed program consists of a finite number of processes and a finite number of channels between the processes. Figure 1 shows an example where each process is represented by a circle and channels are represented by directed edges.

Figure 1. A Distributed System

Processes in a distributed program communicate by sending and receiving messages. Channels are assumed to have infinite buffers, to be error-free and to deliver messages in the order sent. Following are some definitions from [1].

Definitions:

An *event e* is a 5-tuple <*p,s,ss,M,c*> where *p* is a process, *s* and *ss* are *states* of the process before and after the event, *M* and *c* are the *message* and the *channel* through which the message is sent or received by *p* at that event. *M* and *c* can have the special value *null* if no message is involved in the event.

A global state S_r consists of the states of processes of the computation and the states of channels.

We briefly restate Chandy and Lamport's algorithm, which we will call the C&L Algorithm, to record the global state. In that algorithm, each process records its own state, and the two processes upon which a channel is incident cooperate in recording the channel state. The algorithm, which can be initiated independently by more than one process at the same time, is as follows:

C&L Algorithm:

Marker-Sending Rule for a Process p.

For each channel *c*, incident on, and directed away from *p*:

p sends one marker along *c* after *p* records its state and before *p* sends further messages along *c*.

Marker-Receiving Rule for a Process q.

On receiving a marker along a channel *c*:

if *q* has not recorded its state **then**
  q records its state;
  q records the state *c* as the empty sequence
else
  q records the state of *c* as the sequence of messages received along *c* after *q*'s state was recorded and before *q* received the marker along *c*

Theorem 1.

The recorded state S_r is *globally consistent*.

Proof:

See [1]. □

2.2. Halting Algorithm

We now present an algorithm to halt the processes to yield a globally consistent halted state S_h that is equivalent to the recorded state S_r. These states are equivalent in the sense that the state of each process in the halted global state S_h is the same as the state of each process in the recorded global state S_r and the undelivered messages in each channel in S_h are the same as the recorded messages in the state of the channel in S_r. We begin the discussion with the same model as in [1].

2.2.1. Basic Algorithm

Our model is the same as in[1] except that we use a *halt marker* instead of *marker*. This halt marker carries with it a sequence number referred to as *halt_id*. This halt_id enables each process to distinguish an old halt marker (to ignore) from a new halt marker. Each process also keeps track of the latest halt_id received as *last_halt_id* whose value is initially set to zero. Like the C&L Algorithm, halting can be initiated spontaneously by more than one process. The decision as to when to halt can be made independently by each process. We discuss how to set and detect breakpoints in distributed debugging system in section 3.

Halting Algorithm:
  Marker-Sending Rule for a Process p.
    Increment *last_halt_id*;
    Halt Routine (p)
  Marker-Receiving Rule For a Process q.
    On receiving a *halt marker* along a channel c:
      Compare the *halt_id* with its *last_halt_id*;
      if *halt_id* is greater than *last_halt_id* **then**
        Update *last_halt_id*;

Halt Routine (q);
else
Ignore;
Halt Routine (x: process):
For each channel c, incident on and directed away from process x, send a *halt marker* with a *halt_id* equal to the *last_halt_id* along c;
Halt;

A process halts either by receiving halt marker from any one of its adjacent neighboring processes or by spontaneously deciding to halt. If a process halts by receiving a halt marker, it does so on receiving the first halt marker with the new halt_id (old halt markers are left over from previous haltings). When all processes are finally halted, the state of each process is preserved. Each outgoing channel contains undelivered messages with a halt marker as the last one, or is empty if the halt marker was delivered to the receiving process. Given the assumptions of reliable channels and each process observing the same algorithm, it can be shown that when all processes halt, the value of each process's last_halt_id is the same. This is true because the initial value of each last_halt_id is zero and gets incremented exactly once during the Halting Algorithm (since a process can halt only once). The global halted state $S_h$consists of the halted states of the processes and undelivered messages in channels. We claim that S_h is the same as S_r in the sense that

(1) the state of each process in S_h is the same as the recorded state of the corresponding process in S_r; and

(2) the undelivered messages in each channel in S_h are the same as the recorded state of the corresponding channel in S_r.

We begin the proof of our claim with two lemmas.

Lemma 2.1.

The halted state of each process in S_h is the same as the recorded state of the process in S_r.

Proof:

The Halting Algorithm is structurally identical to the C&L Algorithm. In the Halting Algorithm, each process halts at the instant it would record its state in the C&L Algorithm. So the halted state of each process p in S_h is the same as the recorded state of the process in S_r. □

Lemma 2.2.

The undelivered messages in each channel in S_h are the same as the recorded messages of the state of the corresponding channel in S_r.

Proof:

In the Halting Algorithm, a process *p* halts as soon as it receives a *halt marker* on any one of its incoming channels. When a *halt marker* is received on a channel, we know that the channel is now empty since the process that was sending on the channel halted as soon as it sent the *halt marker* on the channel. All of process *p*'s other incoming channels will contain pending messages. Since each process sends a *halt marker* before it halts, the last message in each of these pending channels is the *halt marker*. Therefore, the state of an incoming channel c of a process p in S_h either consists of (zero or more) pending messages followed by a *halt marker* or is empty.

In the C&L Algorithm, each process proceeds with its computation after it records its state when it receives the first *marker* from any of the incoming channels. The state of an incoming channel c of a process p in S_r consists of the sequence of recorded messages received on the channel until a *marker* is received on the channel. Since each process in C&L Algorithm sends a *marker* at the instant it would send a *halt marker* in the Halting Algorithm, the sequence of recorded messages in S_r received on each incoming channel c until a *marker* is received is the same as the stored messages in the channel c in S_h. □

Theorem 2.

S_h is the same as S_r.

Proof:

The proof follows from Lemma 2.1 and Lemma 2.2. □

2.2.2. Problems with the Basic Algorithm

There are two problems with our Halting Algorithm that also occur in the C&L Algorithm. The first problem is how to halt a process that has only infrequent interactions with the other processes of the computation. The process would eventually halt, potentially long after all other processes have halted. Even though nothing is conceptually wrong with this kind of process, it is awkward in practice.

The second problem is one that can make both the Halting Algorithm and the C&L Algorithm fail. This problem occurs when the network connection is acyclic, as in producer-consumer or pipeline relationship. Figure 2 shows an example of this case.

Figure 2. Producer - Consumer Connection

If halting is initiated by the consumer process in this example, there is no way to send the halt marker to the producer process to halt the entire computation. The C&L Algorithm avoids this problem by assuming that the processes are strongly connected.

2.2.3. Extended Model

We now present our model of the interactive distributed debugger system that works with our Halting Algorithm and solves the problems mentioned before. In our extended model, there is an additional process *d* as the debugger process of the system, and there are two additional *control channels* connecting the debugger process with each user process. The introduction of a debugger process solves not only the problems mentioned above but is also a natural structure for an interactive debugging system [3].

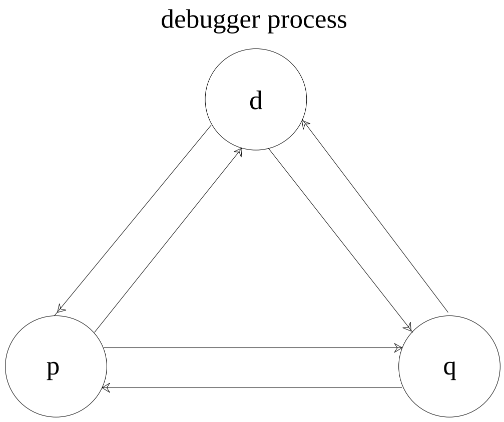

Figure 3. A Distributed System with a Debugger Process

Figure 3 shows the model with user processes *p, q* and debugger process *d*. Since each process has two control channels, one to and one from the debugger process, the network is strongly connected, i.e., there always is a message path from a process to any other process. In addition to guaranteeing strong connectivity of the network, the debugger process performs the typical functions of a debugger. The algorithm to halt the computation need not be changed except that the debugger process *d* never really halts and user processes are always willing to accept a message from the debugger process.

2.2.4. Order of Halting

A process may have more than one incoming channel. This means that a halt marker could be received from any one of the processes attached to these channels, depending on when and from where the halting is initiated. The order in which the processes halt can provide useful information to the programmer, but this information is not available in our Halting Algorithm.

The halting order information can be obtained by making a small change to the halt markers while leaving the structure of the Halting Algorithm unchanged. Each process will append its name to the halt marker before sending the marker to the next process(es). The halt marker that a process receives then describes which processes have already been halted.

3. Distributed Breakpoints

3.1. Types of Breakpoints in Distributed Debugger

In a sequential programming, the decision to halt the program is usually done by specifying predicates about the program's behavior and state. The satisfaction of these predicates corresponds to interesting points in the execution of the program, which we call *breakpoints*. The predicates are expressed in terms of events that correspond to a particular behavior or change in state of the program.

A predicate that is based entirely on the execution behavior or state of a single process is called a *Simple Predicate*. We can combine the Simple Predicates using the disjunctive operator to make a *Disjunctive Predicate*. Likewise, we can combine the Simple Predicates using the conjunctive operator to make a *Conjunctive Predicate*.

Predicates can also be combined to describe a sequence of events. For example, a user may want to halt a program and examine its state when a specified sequence of events is observed during the execution of the program. We call such predicates *Linked Predicates*. Linked Predicates have been used with hardware-based debugging tools such as logic state analyzers. For example, the programmer specifies a non-contiguous sequence of values (such as program addresses) that must occur and the debugging tool detects when this sequence has occurred.

There is usually more than one thread of control in a distributed program and the breakpoint predicates can involve more than one process. We call such predicates *distributed predicates*. We now describe distributed predicates and how to detect the satisfaction of these predicates. When the distributed predicate is satisfied, the Halting Algorithm presented in Section 2 is used to halt the computation.

3.2. Simple Predicates (SP)

Simple Predicates consist of the typical predicates used in sequential program debuggers, such as entering a particular procedure. We also have interprocess event predicates such as a message sent or received, a channel created or destroyed, or a process created or terminated.

3.3. Disjunctive Predicates (DP)

Disjunctive Predicates are specified by expressions using the disjunctive operator ‘‘$\cup$’’:

DP ::= SP [$\cup$ SP]$*$.

The Disjunctive Predicate is satisfied when one or more of the Simple Predicates is satisfied. Halting can be initiated at the instant when any of the SP’s of the DP is satisfied. Multiple SP’s of the DP can be satisfied at the same virtual time. Since the Halting Algorithm works for simultaneous initiations from multiple processes, each process where any SP is satisfied can initiate the Halting Algorithm.

3.4. Linked Predicates (LP)

Linked Predicates specify sequences of events that can be ordered by the happened-before relation and are specified by expressions using the ‘‘$\rightarrow$’’ operator:

LP ::= DP [$\rightarrow$ DP]$*$.

The semantics of LP can be interpreted as follows:

Let Σ be the set of DP_i’s such that $\Sigma = \{DP_i,\ i = 1..n\}$.
Then, the Linked Predicate
$$LP = DP_i \rightarrow DP_j \rightarrow DP_k \ . . . \quad 1 \le i,j,k \le n$$
means the following regular expression
$$LP = DP_i\ [\Sigma - DP_j]*\ DP_j\ [\Sigma - DP_k]*\ DP_k . . .$$

The implementation of the Linked Predicates will be described in section 3.6.

3.5. Conjunctive Predicates (CP)

The Conjunctive Predicates are specified by expressions using the conjunctive operator ‘‘$\cap$’’:

CP ::= SP [$\cap$ SP]$*$.

A Conjunctive Predicate is said to be satisfied at the instant when all the Simple Predicates of the Conjunctive Predicate are satisfied. There is no single time reference across machine boundaries in a distributed

system, so we cannot precisely detect the simultaneous events needed for the Conjunctive Predicate. This form of predicate is well defined within a single machine, but can have several interpretations in a distributed system. Based upon the virtual time concept of a distributed system, our interpretation is as follows. Given two processes P_1 and P_2 residing on different machines, each process has its own virtual time axis, called T_1 and T_2 respectively. Predicate SP_1 is on the state of P_1 and SP_2 on the state of P_2. We define a pair of virtual time points (t_1, t_2) to describe a time when SP_1 is satisfied such that $t_1 \in T_1$ (written as: $SP_1(t_1)$ is *true*), and the time when SP_2 is satisfied such that $t_2 \in T_2$ (written as: $SP_2(t_2)$ is *true*).
We define a set of these virtual time pairs, called SCP, to be:

$$SCP = \{(t_1, t_2) \mid t_1 \in T_1,\ t_2 \in T_2,\ SP_1(t_1) \cap SP_2(t_2)\}.$$

At any point in the set SCP, the conjuntive predicate $SP_1 \cap SP_2$ is satisfied.

Since T_1 and T_2 are virtual time axes, it is not always possible to order a given virtual time t_1 on P_1 and a given virtual time t_2 on P_2 according to Lamport's happened-before relationship. We can divide the SCP into two subsets named $ordered-SCP$ where there is an ordering between t_1 and t_2, and $unordered-SCP$ where there is no ordering. Since the Linked Predicates introduced in the previous section is a mechanism to detect events with ordering, the two subsets can be expressed as follows:

$$ordered-SCP = \{(t_1, t_2) \mid (t_1, t_2) \in \text{SCP},\ ((SP_1)^i \rightarrow (SP_2)^j) \cup ((SP_2)^i \rightarrow (SP_1)^j) \text{ such that } 1 \leq \text{i, j}\}^{\dagger},$$

$$unordered-SCP = \{(t_1, t_2) \mid (t_1, t_2) \in \text{SCP},\ (t_1, t_2) \notin \text{ordered-SCP}\}.$$

Figure 4 shows examples from each of these sets. We see an ordering in time pair (t_{11}, t_{23}) and no ordering possible in (t_{12}, t_{22}).

† We use $(SP)^i$ as a shorthand for $SP \rightarrow SP \cdots \rightarrow SP$. For example, $(SP)^3$ stands for $SP \rightarrow SP \rightarrow SP$.

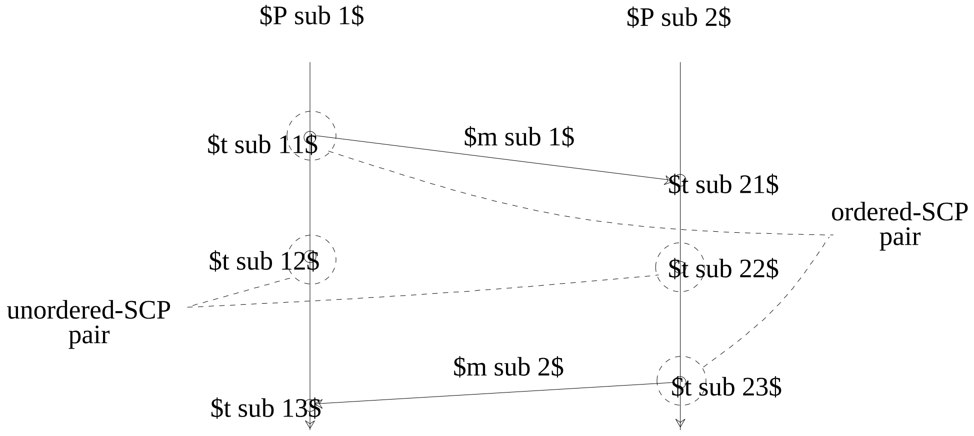

Figure 4. Examples from the Set SCP

We can use the algorithm for detecting satisfaction of Linked Predicates (see section 3.6) for detecting events that occur at virtual times belonging to the set ordered-SCP. For example, if $SP_1(t_{11})$ is *true*, and $SP_2(t_{21})$ and $SP_2(t_{22})$ are *true*, we can use the Linked predicate $SP_1 \rightarrow SP_2$ to detect the events at the time pair (t_{11}, t_{21}). We can use $SP_1 \rightarrow (SP_{21})^2$ to detect the events at the time pair (t_{11}, t_{22}). Halting is initiated at the moment when the last predicate in the ordering is satisfied. Detecting events that occur at virtual times belonging to the unordered-SCP is more difficult. For example, if we detect that SP_1 on P_1 is satisfied, we must also detect SP_2 on process P_2. Since there is no common time reference in a distributed system, it is necessary to have some process gather the information from the other process(es) before halting is to be initiated. We cannot decide until the last notification arrives at the information gathering process, and the inherent time delay in such information gathering makes it impossible for the processes to halt soon enough to preserve the meaningful states of the processes.

3.6. Implementation of Linked Predicate Detection

Since the definition of the Linked Predicate is general enough to comprise the Simple Predicate and the Disjunctive Predicate, only one algorithm is needed to detect the predicates. In addition to the halt marker for the Halting Algorithm, we need a *predicate marker* to carry the Linked Predicate. If necessary, we can append to every message originated by the program some kind of tag so that each process can distinguish the genuine messages from halt markers and predicate markers which are introduced by the debugging system.

To issue the Link Predicate $DP_1 \rightarrow DP_2 \rightarrow DP_3$, the debugger process sends a predicate marker containing the Linked Predicate to each process involved in DP_1. Upon receiving the predicate marker, each process sets up the condition to detect when DP_1 is satisfied. When DP_1 is satisfied at process p, process p creates a new predicate, *newLP*, from the remainder ($DP_2 \rightarrow DP_3$) of the original Linked Predicate. This new predicate is issued to each process involved in DP_2. This process is repeated until last Disjunctive Predicate (in this case, DP_3) in the Linked Predicate is satisfied, at which time a process knows that it should initiate the Halting Algorithm.

Linked Predicate Detection Algorithm:

> *Predicate-Marker-Sending Rule for a process p*.
>
> > Send a *predicate marker* containing the Linked Predicate to each process involved in the first *Disjunctive Predicate* of the *Linked Predicate*;
>
> *Predicate-Marker-Receiving Rule for a process q*.
>
> > On receiving a *predicate marker* from other process:
> >
> > > Separates the first *Disjunctive Predicate* from the *Linked Predicate* carried by the *predicate marker*;
> > >
> > > Make a *newLP* from the received *Linked Predicate* by excluding the first *Disjunctive Predicate*;
> >
> > When the extracted *Disjunctive Predicate* is met:
> >
> > > **if** the *newLP* is *null* **then**
> > >
> > > > Initiate the halting Algorithm;
> > >
> > > **else**
> > >
> > > > Send a new *predicate marker* containing the *newLP* as the new Linked Predicate according to the *Predicate-Marker-Sending Rule*.

Halt markers are manipulated only by the Halting Algorithm and Predicate Markers are manipulated only by the Predicate Detection Algorithm, so these algorithms do not interfere with each other.

4. Application to Current Research

Distributed debugging is an area of active research. For our purposes, we can separate this research into two approaches. The first approach avoids the problem of stopping a program by providing tools only for monitoring a program's execution [3-6]. For example, Bates and Wileden[4] define an Event Description Language (EDL) that allows a programmer to group low-level events into high-level abstract events. EDL requires the ability to observe sequences of events and recognize pattern in these sequences. Our algorithm for recognizing distributed predicates (Section 3.6) could be used to support an EDL abstract event recognizer.

A second approach to distributed debugging is one that more closely approximates traditional, single-process debuggers [7-9]. For example, IDD [8] provides a stepping mode of execution for a collection of processes because IDD does not guarantee that a program can be halted in a timely and consistent manner. The suggested IDD strategy is for a programmer to individually halt processes early enough so that the entire computation is halted before the interesting points are reached. The programmer can then execute the program in single instruction steps to find the error. The Halting Algorithm using distributed breakpoints would simplify the programmer's debugging task.

A variation on the second approach re-routes all normal communications through a centralized debugger process [10, 11]. While this simplifies the detection of distributed breakpoints by providing a single point of event ordering, it also has several disadvantages. First, there can be substantial communication overhead in re-routing the messages through a central hub. Second, the change in message flow could substantially change the execution of the program. Last, the facility to re-route the communications can be complex to build.

The Linked Predicates are similar to Path Expressions [12]. Our distributed predicate detection algorithm provides a vehicle to implement Path Expressions in a distributed system.

5. Conclusion

Interactive debugging is a familiar scenario to any programmer. The Halting Algorithm presented in Section 2 and the definition of distributed breakpoints in Section 3 provide the programmer with the necessary tools to apply these techniques to a distributed program. The fundamental idea is that the program's view of event ordering and relative timing is preserved.

We have presented a definition of breakpoints in a distributed system. This definition shows what type of logical statements make sense in such an environment. A satisfying result is that the types of breakpoints that make sense (those excluding Conjunctive Predicates based on unordered events) are not difficult to implement; the type of breakpoint that is difficult to implement turns out to not be desirable.

Any software debugging tool will cause some change in the absolute timing of a program. We have not tried to avoid this, but our algorithms should only impose a minimal change on the execution of a program. This change is one that should not affect any but the most timing sensitive programs — and for these programs, a hardware monitor may be the only suitable form of debugger.

6. REFERENCES

[1] K. M. Chandy and L. Lamport, ‘‘Distributed Snapshots: Determining Global States of Distributed Systems,’’ *ACM Trans. Computing Systems* **3**(1) pp. 63-75 (February 1985).

[2] L. Lamport, ‘‘Time, Clocks, and the Ordering of Events in a Distributed System,’’ *Communications of the ACM* **21**(7) pp. 558-565 (July 1978).

[3] H. Garcia-Molina, F. Germano, Jr., and W. H. Kohler, ‘‘Debugging a Distributed System,’’ *IEEE Trans. on Software Engineering* **SE-10**(2) pp. 210-219 (March 1984).

[4] P. Bates and J. C. Wileden, ‘‘An Approach to High Level Debugging of Distributed Programs,’’ *Proc. of the SIGSOFT/SIGPLAN Symp. on High-Level Debugging*, pp. 107-111 Pacific Grove, Calif., (August 1983).

[5] R. J. LeBlanc and A. D. Robbins, ‘‘Event-Driven Monitoring of Distributed Programs,’’ *Proc. of the 5th International Conf. on Distributed Computing Systems*, pp. 515-522 Denver, (May 1985).

[6] B. P. Miller, C. Macrander, and S. Sechrest, ‘‘A Distributed Programs Monitor for Berkeley UNIX,’’ *Software - Practice and Experience* **16**(2) pp. 183-200 (February 1986).

[7] E. T. Smith, ‘‘Debugging Techniques for Communicating, Loosely-Coupled Processes,’’ Ph.D. Dissertation - Technical Report TR100, Univ. of Rochester (December 1981).

[8] P. K. Harter, Jr., D. M. Heimbigner, and R. King, ‘‘IDD: An Interative Distributed Debugger,’’ *Proc. of the 5th International Conf. on Distributed Computing Systems*, pp. 498-506 Denver, (May 1985).

[9] F. Baiardi, N. De Francesco, and G. Vaglini, ‘‘Development of a Debugger for a Concurrent Language,’’ *IEEE Trans. on Software Engineering* **SE-12**(4) pp. 547-553 (April 1986).

[10] R. Curtis and L. Wittie, ‘‘BUGNET: A Debugging System for Parallel Programming Environment,’’ *Proc. of the 3rd International Conf. on Distributed Computing Systems*, pp. 394-399 Denver, (August 1982).

[11] R. D. Schiffenbaur, ‘‘Interactive Debugging in a Distributed Programs,’’ *M.S. Thesis*, M.I.T., (August 1981).

[12] B. Bruegge and P. Hibbard, ‘‘Generalized Path Expressions: A High Level Debugging Mechanism,’’ *Proc. of the SIGSOFT/SIGPLAN Symp. on High-Level Debugging*, pp. 34-44 Pacific Grove, Calif., (August 1983).