Distributed Breakpoint Detection in Message-Passing Programs

Lúcia Maria de A. Drummond

Departamento de Computação
Instituto de Matemática
Universidade Federal Fluminense
R. São Paulo, s/n
24040-110 Niterói - RJ
Brazil

Valmir C. Barbosa

Programa de Engenharia de Sistemas e Computação, COPPE
Universidade Federal do Rio de Janeiro
Caixa Postal 68511
21945-970 Rio de Janeiro - RJ
Brazil

Abstract

The ability to set breakpoints stands, along with the possibility of deterministic re-execution, as one of the most important issues in the debugging of message-passing programs. We consider in this paper the design of fully distributed algorithms for the detection of breakpoints in such programs, and provide four algorithms, each for a different type of breakpoint. One of the algorithms detects the occurrence of unconditional breakpoints, while the other three detect the occurrence of breakpoints on disjunctive predicates, stable conjunctive predicates, and generic conjunctive predicates. All the algorithms we present detect breakpoints in the form of earliest global states with respect to the particular property involved. In the case of unconditional breakpoints, such an earliest global state must coincide exactly with the requested local unconditional breakpoints for the processes that do actually participate in the breakpoint. In the case of the other (conditional) breakpoints, what is detected is the earliest global state at which either the disjunctive or conjunctive predicate under consideration is true. In order to actually halt the computation at the exact global state the algorithms detect, we suggest as a first approach the use of checkpointing and rollback-recovery techniques, and indicate for each of the four cases how to approach the recording of checkpoints so that the additional storage requirement per process is not demanding beyond the actual needs.

1. Introduction

The problem we address in this paper is the problem of debugging message-passing programs written for distributed-memory parallel processing machines. This problem has traditionally been treated from two distinct, although complementary, perspectives. The first one concentrates on the ability to deterministically re-execute programs aiming at the exact reproduction of the behavior observed in the previous execution. The second perspective is that of setting breakpoints where the program is to be halted for its context to be examined. Here breakpoints are distributed, comprising a collection of interrelated conditions spread throughout the system. Because of the inherent asynchronism of those machines, and aggravated by the assumed absence of a shared address space, both views of distributed program debugging must of necessity depart significantly from the approaches traditionally adopted in the debugging of sequential programs. Various contributions have appeared in the literature in recent years [1, 4, 6, 8–15, 20, 21, 23–25], including the technical report [6] of which this paper is an expanded and revised version.

Within the broad spectrum of problems associated with the debugging of distributed programs, in this paper we concentrate on the design and analysis of distributed algorithms to detect the occurrence of breakpoints. The breakpoints we consider are of three types. *Unconditional breakpoints* comprise pre-established points specified distributedly among the program's components. In contrast, the *conditional breakpoints* we consider are either *conjunctive* or *disjunctive*. The former specify a global condition as the logical conjunction of a set of local conditions spread throughout the system (a *conjunctive predicate*), while for the latter the global condition is given by the logical disjunction of the local conditions (a *disjunctive predicate*). In the case of conjunctive breakpoints we also consider the particular situation in which the breakpoint reflects a *stable predicate*, that is, a predicate that holds indefinitely further on once it becomes true.

Before we can be more specific about the nature of these breakpoints and the properties of the algorithms we propose in this paper, it is convenient to introduce the details of the computational model we assume in the sequel. The program to be debugged is represented by the undirected graph $G = (N, E)$ with $n = |N|$ and $e = |E|$. For $i = 1, \ldots, n$, $p_i \in N$ is a *process* which may communicate exclusively by message passing with those (and only those) processes $p_j \in N$ such that $(p_i, p_j) \in E$, where $1 \leq j \leq n$. Processes with which p_i may communicate are called its *neighbors*; the set of p_i's neighbors is denoted by N_i. Edges in E represent bidirectional communication channels, which we assume to have infinite capacity (so that processes need never stop executing upon attempting to send a message). We also assume full asynchronism, in the sense that processes are driven by independent, local clocks, and that messages suffer finite, though unpredictable, delays to

be delivered among neighbors.

One helpful abstraction is to model the computation that takes place locally at a process by a sequence of *events*. Associated with an event are then the identification of the process where it takes place (although this identification is not necessarily available to the processes to compute on), the *local time* (as given by the process's local clock) at which the event takes place, as well as possible changes in the *local state* of the process and the sending (receipt) of messages to (from) neighbors. At most one message may be received in association with an event, although no such restriction is placed on the number of messages sent in connection with the event. The distributed computation that takes place over G is then naturally associated with the set of events occurring at all processes, which we henceforth denote by V.

A binary relation on V, here denoted by $\rightarrow$ and usually interpreted as a "happened before" relation is defined as follows [18]. Two consecutive events, say ξ_1 and ξ_2, happening in this order at the same processor, contribute to $\rightarrow$ with the pair (ξ_1, ξ_2). The other pairs in $\rightarrow$ are (ξ_1, ξ_2) such that a message exists which is sent by a process in connection with ξ_1 and received at a neighbor process in connection with ξ_2. The transitive closure $\rightarrow^+$ of $\rightarrow$ is by definition an irreflexive transitive binary relation on V, and constitutes as such a partial order on V. The relation $\rightarrow^+$ is instrumental in the definition of a very important entity in the context of this paper, namely a *consistent global state*, or more succinctly a *global state*. A global state is a partition (V_1, V_2) of V such that $\xi \in V_1$ only if every ξ' such that $\xi' \rightarrow^+ \xi$ is a member of V_1 as well [3]. Clearly, every global state is such that no pair $(\xi, \xi') \in \rightarrow^+$ exists such that $\xi \in V_2$ and $\xi' \in V_1$. Loosely, then, this definition is meant to capture system-wide states that "may actually happen," although this interpretation implicitly refers to a global clock, whose existence is ruled out by assumption.

The problem of detecting a breakpoint and halting the program accordingly should then be posed as the problem of detecting a global state (and then halting the program) at which the condition that specifies the breakpoint holds. Failure to pose the problem like this may lead to impossibilities like, for instance, requiring that a process halt after receiving a certain message, while the sender of that message is required to halt before the message is sent.

Algorithms to solve the breakpoint detection problem should never miss a global state with the required properties if one exists. Under the global-state view of a breakpoint, an unconditional breakpoint is specified for a subset of N as a set of local times, one for each of the participating processes (we assume throughout that this subset has at least one process). If this specification does not constitute a global state, this should be detected by the algorithm and reported as an error. Hereafter, the local time specifying

the unconditional breakpoint at each of the participating processes is called the process's *local unconditional breakpoint* and denoted by lub_i for $p_i \in N$. Conditional breakpoints, in their turn, are specified as a set of *local predicates*, one for each of the participating processes, of which we assume henceforth there is at least one. If no global state exists in which the required conjunctive or disjunctive predicate holds, then the program should simply not halt before it terminates. The local predicate associated with p_i is henceforth regarded as the boolean variable lp_i.

Often the problem, when posed for conditional breakpoints, comes with the additional requirement that the detected global state be the "first" global state for which the predicate (conjunctive or disjunctive) to be watched for holds during the computation. The notion one wishes to capture here is that of the *earliest* global state for which a certain property holds during the computation, and is formalized as follows. Say that a global state (V_1', V_2') is *earlier* than a global state (V_1, V_2) if and only if $V_1' \subset V_1$. Then a global state (V_1, V_2) is an earliest global state for which the property holds if and only if no global state (V_1', V_2') for which the property holds is earlier than (V_1, V_2). It should be clear that this definition allows more than one earliest global state to exist with respect to the same property, for example (V_1, V_2) and (V_1', V_2') such that neither $V_1 \subset V_1'$ nor $V_1' \subset V_1$. If the predicate under consideration is a stable predicate, then the other global states at which it holds in addition to (V_1, V_2) include all the global states (V_1', V_2') such that $V_1 \subset V_1'$.

In the case of unconditional breakpoints, requiring the earliest global state to be detected is only meaningful if there is at least one process that does not participate in the breakpoint. Processes like this have their local times for participation in the breakpoint left "unspecified," and may then be required to appear in the detected global state with as early a local state as possible.

A global state defined as a partition (V_1, V_2) of V is normally viewed as comprising a local state for each process (the state at which the process is left immediately after the occurrence of all pertinent events in V_1) and one set of messages for each channel in each direction (messages sent in connection with events in V_1 to be received in connection with events in V_2). However, all the breakpoints we consider are defined exclusively with respect to the local states of some processes, so a global state may in our context be regarded as comprising n local states only. In addition, we do for simplicity make the assumption that local times always increase at the occurrence of events and remains constant between consecutive events, so that a process's local state is unequivocally determined by the process's local time (local times are then in reality event counters). A global state is then henceforth viewed as an n-component array of local times. If φ is a global state and $lt_i \geq 0$ denotes (as it does henceforth) p_i's local time for $p_i \in N$, then the component of φ

corresponding to p_i is $\varphi[i] = lt_i$ for the appropriate value of lt_i.

Under this view of a global state, an n-component array φ of local times is a global state if and only if no $p_i \in N$ exists that ever receives a message earlier than (or at) $\varphi[i]$ which was sent by some $p_j \in N_i$ later than $\varphi[j]$. It should also be noted, still in connection with this view of a global state, that a global state φ is an earliest global state for which a certain property holds if and only if no other global state φ' for which the property holds is such that $\varphi'[k] \leq \varphi[k]$ for all $p_k \in N$.

Distributed algorithms for breakpoint detection are evaluated with respect to the overhead incurred by the computation being monitored when they run. Following the customary complexity measures adopted for asynchronous distributed algorithms, this overhead is given as asymptotic worst-case expressions for the number of additional message bits passed between neighbors and the additional time for completion. (Accounting for the number of messages instead of bits is in most cases an equivalent approach. In this paper, however, additional communication is often incurred in the form of additional fields attached to messages of the computation proper, and this would not show if we adopted message counts.)

While counting the number of message bits to obtain the *message complexity* is in most cases a simple matter, the system's inherent asynchronism requires a little more elaboration for the definition of the *time complexity*. Normally local computation is assumed to take no time, and then the time complexity expresses the longest chain of the type "receive a message and send a message as as consequence" occurring during the computation [19]. However, in reality such a causal chain is interspersed with the occurrence of events unrelated to messages, which at times may be significant, and then the assumption that local computation takes no time becomes erroneous. Our convention in this paper is then to express the time complexity as a pair of measures, one being the traditional measure, to which events unrelated to messages are assumed not to contribute (called *global time complexity* to evidence its system-wide significance), and another to account, within a single process, for the causal chains involving events unrelated to messages only (called *local time complexity*).

Often we will need to refer to the message complexity of the computation proper, which we shall henceforth assume to be $O\big(c(n,e)\big)$ messages, or more succinctly $O(c)$ messages. Likewise, we shall assume that every process's local clock can only represent local times up to a value T (i.e., $lt_i \leq T$ for all $p_i \in N$), which can be arbitrarily large but needs nevertheless be such that we can refer to it when computing our algorithms' complexities.

Following a discussion of related work in Section 2 and some preliminary remarks and

methodological issues in Section 3, in this paper we present four algorithms for breakpoint detection. The first (and simplest) algorithm detects breakpoints on disjunctive predicates and is presented in Section 3 as algorithm DETECT_DP. This algorithm has a message complexity of $O(cn \log T)$ bits, a global time complexity of $O(1)$, and a local time complexity of $O(n)$ per message reception. In addition, $O(n \log T)$ memory bits are required for the maintenance of data structures per process.

In Section 4, algorithm DETECT_UBP for the detection of unconditional breakpoints is introduced. It has a message complexity of $O\big((c + ne)n \log T\big)$ bits, a global time complexity of $O(n)$, and a local time complexity of $O(n)$ per message reception. Its memory requirement per process is $O(n \log T)$ bits.

Algorithm DETECT_STABLE_CP is presented in Section 5 for the detection of breakpoints on stable conjunctive predicates. Its message and time complexities are the same as algorithm DETECT_UBP's, and so is also its memory requirement.

Our last algorithm is given in Section 6 for the detection of breakpoints on generic conjunctive predicates. This algorithm is called DETECT_CP, and has a message complexity of $O\big((c + Pne)n \log T\big)$ bits, where P is the maximum number of times any local predicate becomes true. The global and local time complexities of algorithm DETECT_CP are the same as the previous two algorithms', and so is its memory requirement per process.

Every one of our algorithms is fully distributed, and the detection-related processing performed in connection with each process in N is exactly the same throughout all of N. In particular, no special processes are needed to which information originating from all over the system would be conveyed in order for those processes to perform the detection. Furthermore, what our algorithms detect are earliest global states in which the conditions required for detection hold, and they need no trace of a previous execution to work on. In at least one of these three aspects, all the algorithms we provide are the first to appear in the literature.

Another important characteristic of all four algorithms is that not every process is required to participate in the condition specifying the breakpoint to be detected. Processes that do not participate in the breakpoint do nevertheless participate in the detection. In the case of unconditional breakpoints (which do not appear to have been treated in the literature at all), the algorithm we provide is in addition capable of detecting errors in the specification of the breakpoint.

Because our algorithms perform the detection of earliest global states in which the appropriate conditions hold, a system-wide dissemination of a halt order is needed for every process to stop after the detection. However, in order to halt the computation at the detected breakpoint a checkpointing and rollback-recovery mechanism is needed whereby

every process is brought back to the local state with which it participated in the detected global state. A first approach to this issue is discussed in Section 7, where we also discuss strategies to save memory while maintaining the checkpoints.

In addition to these sections, this paper comprises an additional section with concluding remarks (Section 8) and two appendices. Appendix 1 contains a summary of the notation utilized globally in the paper. Proofs of all the formal results stated throughout the paper are collected in Appendix 2.

2. Related Work

Extensive though the literature on the debugging of distributed parallel programs is becoming, very few authors have addressed the design of distributed algorithms to detect the types of breakpoints we have dealt with in this paper. However, the works of Miller and Choi [24] and of Manabe and Imase [21] deserve special mention for their proximity with the topics we have considered.

Miller and Choi's algorithms [24] are centered around a procedure to spread a halting instruction among the processes of the system, which emanates from a process deciding to halt the computation and reaches the others through a "snapshot"-type algorithm [3]. Based on this procedure, a distributed algorithm is given to detect breakpoints on disjunctive predicates (without however any consideration as to whether the breakpoint detected is the earliest possible), and attempts are also made toward algorithms to detect breakpoints on what the authors call "linked" predicates and on conjunctive predicates that can be expressed as the former.

The issue of detecting earliest global states appears to have been first brought to the fore by Manabe and Imase [21], who gave distributed algorithms to halt a distributed computation at the earliest global state at which a conjunctive predicate holds and to exhibit the value of an expression calculated at the earliest global state at which a disjunctive predicate holds. Both algorithms rely on the "replay" of a distributed computation based on a trace of its execution, much in the style introduced by LeBlanc and Mellor-Crummey [20]. In the conjunctive-predicate case, for example, the trace is used so that processes can poll other processes from which messages are known to be due to arrive. Processes whose local predicates have become true only send messages when thus requested to, and this is what ensures the detection of earliest global states.

Other authors have addressed questions related to the ones that motivated this paper, although with considerably less similarity than the ones we just examined. For example, Spezialetti [28] has proposed a "semi-centralized" approach to detect breakpoints based on Spezialetti and Kearns's [29] concept of "simultaneous regions." Because such regions miss

many of the possible global states in a distributed computation, their approach can only be applied to breakpoints on stable predicates, although without the assurance of finding earliest global states.

Another related question was addressed by Cooper and Marzullo [5], who proposed centralized algorithms to analyze global-state "lattices" in the search for properties that hold across the board from one execution of a distributed program to another. For example, they have an algorithm to detect whether an execution exists in which a given predicate holds at some global state. In the same vein, another algorithm is proposed to detect whether the predicate holds at a global state in every execution. Clearly, the generality of considering multiple executions comes at the price of exceedingly large storage requirements, as the number of possible global states may grow exponentially with the system's size for some computations.

Another centralized approach was recently proposed by Garg and Waldecker [10, 11]. What they have proposed is to employ a separate process (or to endow one of the system's processes with special characteristics) to which information originating at all the other processes is sent. Based on a manipulation of the information that is received, breakpoints on generic conjunctive predicates can be detected by that process. The authors also claim that this detection can be distributed by dividing the processes into hierarchically organized groups. In the algorithm that they present, one process serves as a checker and all other processes keep their own arrays of local times. Whenever the local predicate of a process becomes true for the first time since it last sent a message, it generates a debug message containing its array and sends it to the checker, which then checks whether the conjunctive predicate is satisfied. The process is not required to send its array every time the local predicate is detected, as it suffices to do it only after messages are sent. The checker has a separate queue for each process involved in the predicate. Incoming debug messages from those processes are enqueued in the appropriate queue. The task of the checker is to check the ordering among the arrays. For the conjunctive predicate to be satisfied, the checker must find a set of arrays, one from each queue, such that each is "incomparable" to all others in the set.

3. Preliminaries

Each of the distributed algorithms we introduce in this paper is viewed as being represented by another set of n processes, called $q_1, \ldots, q_n$, each executing mutually exclusively with its counterpart among the n processes of the computation proper that constitute N. Processes $q_1, \ldots, q_n$ communicate with one another by means of messages sent over the communication channels corresponding to the edges in E as well, and are endowed with

the following distinctive abilities for $1 \leq i \leq n$.

(i) Every message incoming to or outgoing from p_i is intercepted by q_i, which may attach new fields to the message or strip some fields off the message before passing it on to p_i or sending it out to some of p_i's neighbors.

(ii) Process q_i is activated (and then p_i is suspended, while lt_i remains constant) when lt_i becomes equal to lub_i (in the case of unconditional breakpoints) or upon the occurrence of a change in the value of the local predicate lp_i (in the case of conditional breakpoints), or yet upon the sending by p_i of a message or the arrival of a message from q_j such that $p_j \in N_i$.

(iii) Process q_i may suspend the execution of p_i at any time, as well as have it resumed (possibly after having altered its context).

Each pair of processes p_i and q_i can then be viewed as sharing a single processor, which is switched between the two for execution. Process p_i takes control only when q_i is not running, unless it has been explicitly suspended by q_i. For simplicity, when necessary we henceforth refer to such a pair of processes as a *node*. Also, henceforth N' denotes the set of processes q_i such that $p_i \in N$ and N'_i the set of processes q_j such that $p_j \in N_i$.

Our algorithms for breakpoint detection are based on the following general approach. For $1 \leq i \leq n$, process q_i maintains an array gs_i of length n representing its view of the global state to be detected, similarly to the "time vectors" treated in [7, 22] and to the basic representations adopted in other approaches related to ours (as those mentioned in Section 2). This array is initialized with zeroes (representing the earliest global state of the computation) and is updated when information is received concerning the other nodes' local unconditional breakpoints or local predicates. Such information is conveyed from node to node either by means of special *broadcast*-type messages or as additional fields attached to the *computation*-type messages that constitute the communication traffic of the computation proper. This information, when sent by q_i, comprises the array gs_i, and may, depending on the type of breakpoint to be detected, comprise additional data as well. In each of the cases we consider, this information is transported among nodes in such a way as to allow at least one node, say the one comprising some $q_k \in N'$, to detect locally that the breakpoint has occurred at the global state recorded in its current gs_k, which is in all cases the earliest global state at which the breakpoint can be said to have occurred. Once the breakpoint is detected, additional measures may be taken to halt the program and then roll it back to the appropriate global state as discussed in Section 7.

Before we proceed, let us pause for a moment and examine algorithm DETECT_DP for the detection of breakpoints on disjunctive predicates. Such a breakpoint is a global

state at which for at least one of the participating processes the local predicate holds. Clearly, the earliest global state at which a disjunctive predicate holds does not have to be unique, as illustrated in Figure 1, so it is conceivable that more than one process detects the occurrence of the breakpoint, however at different global states.

In Figure 1, and in the figures to appear in the remainder of the paper, processes are represented by local-time axes, in which the local times are to be regarded as increasing from left to right. On each axis, thick bars are used to indicate the periods during which the local predicates are true. Arrows represent messages. In the case of Figure 1, there are clearly two earliest global states at which at least one of lp_i or lp_j is **true**.

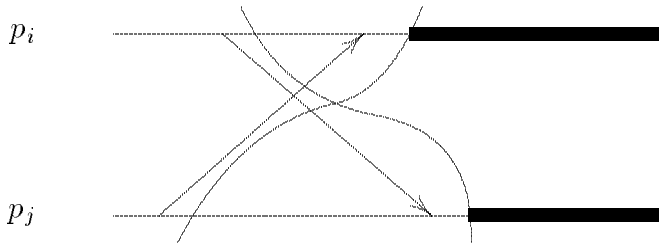


Figure 1. Two earliest global states at which the disjunctive predicate holds

Because of the inherent ease with which disjunctive predicates can be detected in a distributed fashion, algorithm DETECT_DP is quite straightforward. It does not employ any *broadcast* messages, and attaches the array $gs_i(lt_i)$, in addition to a "status bit" (to be discussed shortly), to the *computation* messages sent by $q_i \in N'$. This array is identical to gs_i in all components except the ith, which is given by lt_i. Our earlier assumptions imply that the value of lt_i is in this case the local time at p_i when it sent the message that q_i intercepted, and then corresponds to p_i's local state immediately succeeding the sending of the message. The *computation* messages sent by q_i are then triples like $\big(\text{"status bit"}, gs_i(lt_i), body\big)$, where *body* is the content of the message received from p_i by q_i, denoted by $(body)$. Likewise, when q_i receives a message $(\text{"status bit"}, gs_j, body)$ from $q_j \in N_i'$, it is $(body)$ that gets forwarded to p_i, so that the processes in N only get involved with messages of the computation proper.

Attaching the modified gs_i to *computation* messages is a procedure with important properties in the context of this paper, not only for the algorithm we are beginning to present, but also for other algorithms presented in the sequel. We then pause briefly to introduce the following two supporting lemmas. They will only be used formally in

Appendix 2, but are sufficiently simple that the reader can appreciate their significance in the design of the algorithms to come (this is why we present them at this early stage).

Lemma 1. For all $p_i \in N$, if gs_i is a global state such that $gs_i[i] < lt_i$ and no message is received at p_i at time t such that $gs_i[i] < t \leq lt_i$, then $gs_i(lt_i)$ is also a global state.

Proof: See Appendix 2. ■

Lemma 2. If φ and φ' are global states, then the component-wise maximum of the two is also a global state.

Proof: See Appendix 2. ■

The essence of algorithm DETECT_DP is the following for $p_i \in N$. Variable lp_i is initialized with **false**, and is assumed never to become **true** if p_i does not participate in the breakpoint. Whenever q_i detects that lp_i has become **true**, it sets $gs_i[i]$ to lt_i and declares the breakpoint on the disjunctive predicate detected at the global state gs_i. Because every message it received from $q_j \in N_i'$ prior to lt_i carried a copy of q_j's view of the global state with jth component updated to the time the message was sent, gs_i must indeed be a global state by Lemmas 1 and 2. In order to ensure that it is also an earliest global state with respect to the disjunctive predicate, the simple procedure we just described must only be allowed to be performed if no other process has already detected a global state that renders the one q_i would detect not an earliest one. This is where the "status bit" comes in. This bit will indicate, upon arriving along with a *computation* message, whether any other such global state has already been detected.

Algorithm DETECT_DP is presented next as a set of actions to be performed by q_i. Two additional variables employed by the algorithm are the booleans $found_i$ and $found\_elsewhere_i$, both initially set to **false**, which indicate respectively whether q_i has detected the breakpoint on the disjunctive predicate and whether such a breakpoint has already been detected elsewhere so that the one detected by q_i would necessarily not be an earliest one.

Actions at q_i for algorithm DETECT_DP:

(1) Upon detecting that lp_i has become **true**:

```
if not (found_i or found_elsewhere_i) then
    begin
        gs_i[i] := lt_i;
        found_i := true
    end;
```

(2) Upon receipt of $(body)$ from p_i destined to $p_j \in N_i$:

Forward $(found_i$ **or** $found\_elsewhere_i, gs_i(lt_i), body)$ to q_j;

(3) Upon receipt of $(bit_j, gs_j, body)$ from $q_j \in N_i'$:

$found\_elsewhere_i := bit_j$ **or** $found\_elsewhere_i$;
if not $(found_i$ **or** $found\_elsewhere_i)$ **then**
    for $k := 1$ **to** n **do**
        if $gs_i[k] < gs_j[k]$ **then**
            $gs_i[k] := gs_j[k]$;
Forward $(body)$ to p_i;

Theorems 3 and 4 given next establish, respectively, the correctness and complexity of algorithm DETECT_DP.

Theorem 3. There exist $p_i \in N$ and $t \geq 0$ such that the following three conditions are equivalent to one another for algorithm DETECT_DP.

(a) There exists a global state φ such that $lp_k =$ **true** at time $\varphi[k]$ for at least one $p_k \in N$;

(b) $found_i$ becomes **true** at time $lt_i = t$;

(c) At time $lt_i = t$, gs_i is the earliest global state at which $lp_k =$ **true** for at least one $p_k \in N$.

Proof: See Appendix 2. ■

Theorem 4. Algorithm DETECT_DP has message complexity of $O(cn \log T)$ bits, global time complexity of $O(1)$, local time complexity of $O(n)$ per message reception, and requires $O(n \log T)$ bits of storage per process.

Proof: See Appendix 2. ■

Detecting the other types of breakpoints we consider in this paper is a considerably more intricate task in comparison with the detection of breakpoints on disjunctive predicates. These other cases comprise unconditional breakpoints and breakpoints on conjunctive predicates (both in the stable case and in the general case), all of which require some sort of additional "global" information to be monitored. It is the propagation of this global information that makes use of the *broadcast* messages we introduced earlier.

In general, in addition to gs_i process $q_i \in N'$ also maintains another array of booleans with its local view of the global condition to be monitored and detected. When disseminated by q_i, this array is always accompanied by gs_i as well, so that whenever q_i detects

locally that the global condition has occurred (by examination of its array), it also associates the contents of gs_i with the global state at which the condition occurred.

Broadcast messages are sent by q_i whenever p_i is one of the processes participating in the global condition to be detected and either its local unconditional breakpoint is reached (in the case of unconditional breakpoint detection) or its local predicate becomes true (in the case of the detection of breakpoints on conjunctive predicates). The broadcast we employ is of the "flooding" type, that is, information is sent by q_i to every $q_j \in N_i'$, and so forth until it reaches all nodes. During this propagation of information, an arriving gs_j from some $q_j \in N_i'$ is used by q_i to update gs_i. In addition, gs_j and the other array accompanying it are used to update the local view at q_i of the global condition being monitored.

Some precautions are of course needed in addition to this simple propagation procedure, such as never sending to a process the exact same information received from it, so that the dissemination of information can be guaranteed to terminate. In addition, we adopt a "forward-when-true" rule for the propagation of information. This rule states that a node participates in the broadcast (i.e., forwards the information it receives) only when its local condition (local unconditional breakpoint reached or local predicate become true) holds. Clearly, if no messages were ever sent during the computation proper, then this broadcast would suffice for the detection of the desired type of breakpoint. In such a case, whichever node produced an array with **true** values for all the participating processes would declare the breakpoint detected at the global state given by the global-state array obtained along with it.

Algorithm BROADCAST_WHEN_TRUE does this detection in the absence of messages related to the computation, so long as the global condition under monitoring is stable. In this algorithm, process q_i maintains a boolean variable lc_i to indicate whether the local condition with which p_i participates (if at all) in the global condition to be detected is true. It is initialized with **false** if p_i does indeed participate in the global condition, or with **true** otherwise. Stability then means that no $p_k \in N$ exists such that lc_k is reset to **false** once it becomes **true**. The array associated with q_i's view of the global condition is denoted by gc_i. For $1 \leq k \leq n$, $gc_i[k]$ is initialized with the same value assigned initially to lc_k. Only *broadcast* messages are employed in this algorithm (as the computation proper does not employ any), denoted by the pair (gc_i, gs_i) when q_i is the sender. As in the case of algorithm DETECT_DP discussed earlier, a boolean variable $found_i$, set to **false** initially, is employed to indicate whether q_i has detected the occurrence of the global condition. In addition, another boolean variable, $changed_i$, is used by q_i to ensure that a *broadcast* message is never sent to a node if not different than the last message sent to that node.

Actions at q_i for algorithm BROADCAST_WHEN_TRUE**:**

(1) Upon detecting that lc_i has become **true**:

```
gc_i[i] := lc_i;
gs_i[i] := lt_i;
if gc_i[1] ∧ ··· ∧ gc_i[n] then
    found_i := true
else
    Send (gc_i, gs_i) to every q_k ∈ N_i';
```

(2) Upon receipt of (gc_j, gs_j) from $q_j \in N_i'$:

```
if not found_i then
    begin
        changed_i := false;
        for k := 1 to n do
            if gs_i[k] < gs_j[k] then
                begin
                    gs_i[k] := gs_j[k];
                    gc_i[k] := gc_j[k];
                    changed_i := true
                end;
        if lc_i and changed_i then
            if gc_i[1] ∧ ··· ∧ gc_i[n] then
                found_i := true
            else
                Send (gc_i, gs_i) to every q_k ∈ N_i'
    end;
```

The following two theorems are related to properties of algorithm BROADCAST_WHEN_TRUE.

Theorem 5. There exist $p_i \in N$ and $t \geq 0$ such that the following three conditions are equivalent to one another for algorithm BROADCAST_WHEN_TRUE.

(a) There exists a global state φ such that $lc_k =$ **true** at time $\varphi[k]$ for all $p_k \in N$;

(b) $found_i$ becomes **true** at time $lt_i = t$;

(c) At time $lt_i = t$, gs_i is the earliest global state at which $lc_k =$ **true** for all $p_k \in N$.

Proof: See Appendix 2. ■

Theorem 6. Algorithm BROADCAST_WHEN_TRUE has message complexity of $O(n^2 e \log T)$ bits, global time complexity of $O(n)$, local time complexity of $O(n)$ per message reception, and requires $O(n \log T)$ bits of storage per process.

Proof: See Appendix 2. ■

Algorithms DETECT_DP and BROADCAST_WHEN_TRUE detect breakpoints in two extreme situations, respectively when the breakpoint is on a disjunctive predicate and when the breakpoint is on a conjunctive predicate but the computation proper does not ever send any message (it is simple to note that the case of unconditional breakpoints in the absence of *computation* messages is in fact a case of breakpoints on conjunctive predicates). In the former case only *computation* messages are employed, whereas in the latter case only *broadcast* messages are needed. Other situations between these two extremes are examined in the sequel, and then the messages involved become 4-tuples of the type $(computation, gc_i, gs_i, body)$ for *computation* messages sent by q_i, and $(broadcast, gc_i, gs_i, \textbf{nil})$ for *broadcast* messages sent by q_i. The **nil** field is only used to produce messages with the same number of fields, with the leading type field used to differentiate between message types.

4. Unconditional Breakpoints

In this section we introduce DETECT_UBP, a distributed algorithm to detect the occurrence in a distributed computation of an unconditional breakpoint. As we discussed previously, this unconditional breakpoint is specified, for each process actually participating in the breakpoint, as a local time denoted by lub_i for $p_i \in N$. For processes p_i that do not participate in the breakpoint, we have chosen to adopt $lub_i = \infty$, so that lt_i can never equal lub_i.

Algorithm DETECT_UBP can be regarded as a mixture of algorithms DETECT_DP and BROADCAST_WHEN_TRUE, discussed in the previous section, as it must operate somewhere in-between the two extreme situations assumed by those algorithms. Put differently, the detection of unconditional breakpoints does require the detection of a global condition (which is ruled out by algorithm DETECT_DP) and must be applicable to the case when messages of the computation proper exist (which are disallowed by algorithm BROADCAST_WHEN_TRUE).

The variables employed by algorithm DETECT_UBP are essentially the ones introduced in Section 3 for the other two algorithms, except that for $q_i \in N'$ the boolean variable lc_i is now replaced with the occurrence of the equality $lt_i = lub_i$, and furthermore the array ub_i, used to indicate q_i's view of the occurrence of the local unconditional breakpoints at all processes, is now used in lieu of the array gc_i. For $p_k \in N$, $ub_i[k]$ may be either **true**, **false**, or **undefined**. It is **true** or **false** if p_k participates in the unconditional breakpoint and is viewed at p_i as having already reached its local unconditional breakpoint or not, respectively, and is **undefined** if p_k is not one of the processes participating in the

unconditional breakpoint. Initially, $ub_i[k]$ is set to **false** for every participating p_k and to **undefined** if p_k does not participate.

Algorithm DETECT_UBP proceeds as follows. Whenever q_i detects that $lt_i = lub_i$, it updates $ub_i[i]$ and $gs_i[i]$ accordingly and starts a broadcast to disseminate the updated ub_i and gs_i. This broadcast proceeds like the one in algorithm BROADCAST_WHEN_TRUE, i.e., it is never forwarded by a node whose local unconditional breakpoint has not yet been reached (unless the node does not participate in the unconditional breakpoint), and in addition no duplicate information is ever forwarded by any node. *Computation* messages are always sent with ub_i and $gs_i(lt_i)$ attached to them, in the way of algorithm DETECT_DP, so that the global state that is eventually detected is indeed a global state. This detection, if achieved by q_i, corresponds to the verification that $ub_i[k] \neq$ **false** for all $p_k \in N$, that is, every process has either reached its local unconditional breakpoint or is not participating in the unconditional breakpoint.

One of the difficulties in designing algorithm DETECT_UBP is that it must be able to detect situations in which the requested set of local unconditional breakpoints does not constitute a global state. In such situations, an error must be reported and the computation proper must be allowed to progress normally. The detection of such a situation can be achieved along the following lines. Suppose q_i receives a *computation* message, along with the attached ub_j and gs_j, from some $q_j \in N_i'$. If $ub_j[j] =$ **true** and $ub_i[i] =$ **false** at this moment, then clearly an error has occurred in the determination of the unconditional breakpoint, as p_i will never reach its local unconditional breakpoint in such a way that is consistent with the local unconditional breakpoint of p_j from the point of view of a global state. This is illustrated in Figure 2, in which the partition of the event set indicated by the dashed line cannot possibly be a global state.

The possibility of having nodes for which no local unconditional breakpoint is specified complicates the treatment of these erroneous conditions a little bit. If a causal chain of *computation* messages beginning at q_ℓ such that $ub_\ell[\ell] =$ **true** and going through a number of nodes q_k for which $ub_k[k] =$ **undefined** eventually leads to q_i such that $ub_i[i] =$ **false**, then an error must be detected just as in the case discussed earlier. The way we approach this is by artificially setting $ub_k[k]$ to **true** for all the q_k's. A boolean variable in_error_i, initially set to **false**, is employed by q_i to indicate whether an erroneous condition has been detected.

Nodes that do not participate in the unconditional breakpoint also complicate the detection of earliest global states. If such nodes did not exist, or if we did not require the earliest global state to be detected when they did exist, then what we have outlined so far would suffice for algorithm DETECT_UBP to work as needed. However, the existence of

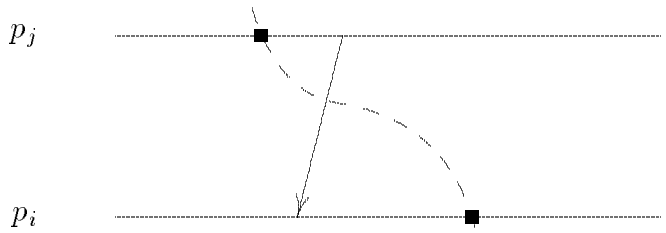
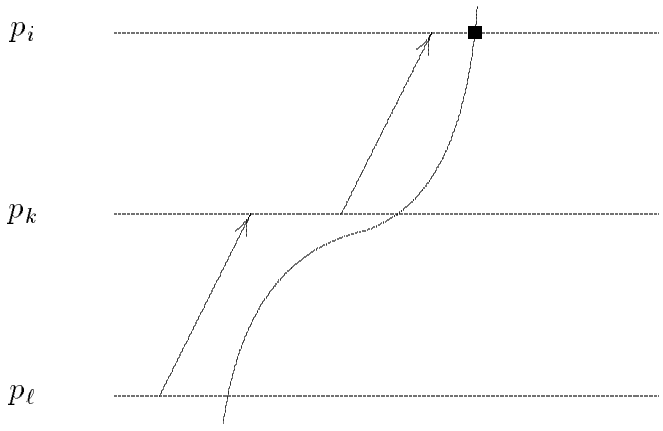
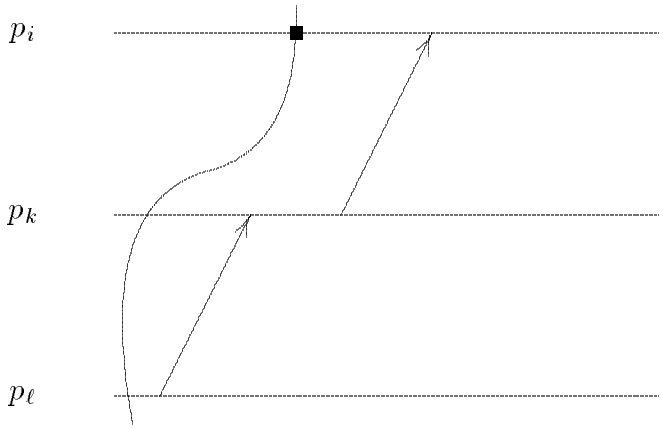


Figure 2. Error in setting local unconditional breakpoints

causal chains of *computation* messages similar to the one we just described but beginning at q_ℓ such that $ub_\ell[\ell] =$ **undefined** may lead to distinct earliest global states, depending on whether it leads to q_i such that $ub_i[i] =$ **false** or $ub_i[i] =$ **true**, as illustrated in Figure 3, whose parts (a) and (b) depict, respectively, the two cases. Only in the former case should q_i take into account what it receives attached to the *computation* message in updating gs_i, but the senders of the preceding messages in the causal chain have no way of knowing this beforehand. The strategy we adopt to tackle this is the following. In addition to maintaining gs_i as a local view of the global state to be detected, q_i also maintains an alternative view, denoted by alt_gs_i, which is initialized like gs_i but only updated or attached to outgoing *computation* messages (the latter in place of gs_i) if $ub_i[i] =$ **undefined**. Arriving *computation* messages at q_i affect gs_i if $ub_i[i] =$ **false** or alt_gs_i if $ub_i[i] =$ **undefined**. So for q_i such that $ub_i[i] =$ **undefined**, $gs_i[k] \leq alt\_gs_i[k]$ for all $p_k \in N$, and therefore gs_i may constitute an earlier global state than alt_gs_i.



(a)



(b)

Figure 3. Earliest global states and unconditional breakpoints

Actions at q_i for algorithm DETECT_UBP**:**

(1) Upon detecting that $lt_i = lub_i$:

if not in_error_i **then**
begin
$ub_i[i] :=$ **true**;
$gs_i[i] := lt_i$;
if $ub_i[k] \neq$ **false** for all $k = 1, \ldots, n$ **then**
$found_i :=$ **true**
else
Send $(broadcast, ub_i, gs_i,$ **nil**$)$ to every $q_k \in N_i'$
end;

(2) Upon receipt of $(broadcast, ub_j, gs_j,$ **nil**$)$ from $q_j \in N_i'$:

if not $(in\_error_i$ **or** $found_i)$ **then**
begin
$changed_i :=$ **false**;
for $k := 1$ **to** n **do**
if $gs_i[k] < gs_j[k]$ **then**
begin
$gs_i[k] := gs_j[k]$;
$ub_i[k] := ub_j[k]$;
$changed_i :=$ **true**
end;
if $ub_i[i] =$ **undefined then**
for $k := 1$ **to** n **do**
if $alt\_gs_i[k] < gs_j[k]$ **then**
$alt\_gs_i[k] := gs_j[k]$;
if $(lub_i \neq$ **false**$)$ **and** $changed_i$ **then**
if $ub_i[k] \neq$ **false** for all $k = 1, \ldots, n$ **then**
$found_i :=$ **true**
else
Send $(broadcast, ub_i, gs_i,$ **nil**$)$ to every $q_k \in N_i'$
end;

(3) Upon receipt of $(body)$ from p_i destined to $p_j \in N_i$:

if $ub_i[i] =$ **undefined then**
Forward $\big(computation, ub_i, alt\_gs_i(lt_i), body\big)$ to q_j
else
Forward $\big(computation, ub_i, gs_i(lt_i), body\big)$ to q_j;

(4) Upon receipt of $(computation, ub_j, gs_j, body)$ from $q_j \in N_i'$:

```
if not (in_error_i or found_i) then
    begin
        if (ub_j[j] = true) and (ub_i[i] = false) then
            in_error_i := true;
        if (ub_j[j] = true) and (ub_i[i] = undefined) then
            ub_i[i] := true;
        if (ub_j[j] = undefined) and (ub_i[i] = false) then
            for k := 1 to n do
                if gs_i[k] < gs_j[k] then
                    begin
                        gs_i[k] := gs_j[k];
                        ub_i[k] := ub_j[k]
                    end;
        if (ub_j[j] = undefined) and (ub_i[i] = undefined) then
            for k := 1 to n do
                if alt_gs_i[k] < gs_j[k] then
                    alt_gs_i[k] := gs_j[k]
    end;
Forward (body) to p_i;
```

Next we give properties of algorithm DETECT_UBP related to its correctness and complexity.

Theorem 7. There exist $p_i \in N$ and $t \geq 0$ such that the following four conditions are equivalent to one another for algorithm DETECT_UBP.

(a) There exists a global state φ such that $\varphi[k] = lub_k$ for every $p_k \in N$ such that $lub_k < \infty$;

(b) in_error_k never becomes **true** for any $p_k \in N$;

(c) $found_i$ becomes **true** at time $lt_i = t$;

(d) At time $lt_i = t$, gs_i is the earliest global state at which $gs_i[k] = lub_k$ for every $p_k \in N$ such that $lub_k < \infty$.

Proof: See Appendix 2. ■

Theorem 8. Algorithm DETECT_UBP has message complexity of $O((c+ne)n \log T)$ bits, global time complexity of $O(n)$, local time complexity of $O(n)$ per message reception, and requires $O(n \log T)$ bits of storage per process.

Proof: See Appendix 2. ■

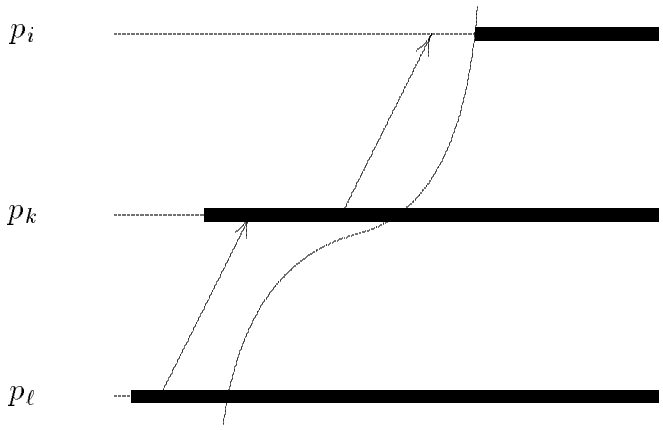
5. Breakpoints on Stable Conjunctive Predicates

In this section we discuss algorithm DETECT_STABLE_CP for the detection of breakpoints on stable conjunctive predicates. Such predicates are specified for each participating process $p_i \in N$ as the local predicate lp_i endowed with the property that it remains true once it becomes true. Unconditional breakpoints are also breakpoints on stable conjunctive predicates, but much more rigid than the ones we consider in this section, as in that case the detected global state is required to match the local unconditional breakpoints specified for the participating processes exactly. In contrast, the ones we are now beginning to consider only ask that the local predicates of the participating processes be true in the detected global state, although in some processes they may have become true earlier than the local times given by the global state. Not surprisingly, then, the algorithm introduced in this section can be regarded as a slight simplification of algorithm DETECT_UBP, as error conditions no longer need to be addressed.

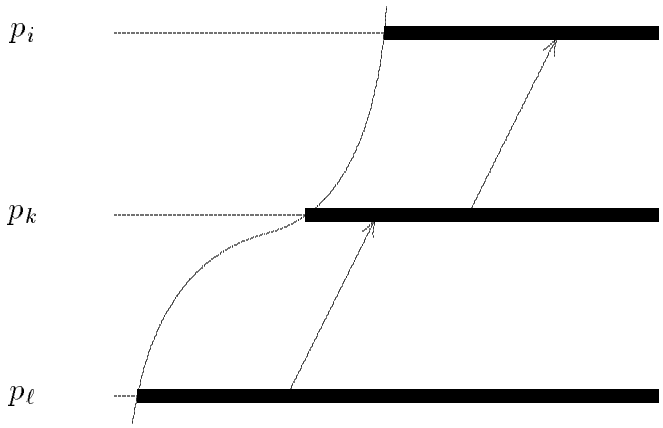
Being in many senses related to algorithm DETECT_UBP, algorithm DETECT_STABLE_CP can also be viewed as a conceptual mixture of the principles employed in algorithms DETECT_DP and BROADCAST_WHEN_TRUE. With respect to the latter, the local condition for $p_i \in N$, lc_i, is now expressed by the very local predicate lp_i we have been considering throughout, and q_i's view of the global condition, gc_i, is now the array cp_i. For all $p_k \in N$, $cp_i[k]$ is initialized like lp_k, that is, to **false** if p_k is participating in the breakpoint, and to **true** otherwise. All the other variables employed by algorithm DETECT_STABLE_CP have the same meaning they had when used in previous contexts.

The simplification of algorithm DETECT_UBP to yield DETECT_STABLE_CP does not go any further than the elimination of error detection, as an alternative local view at q_i of the global state to be detected, alt_gs_i, is still needed to aid in the detection of the earliest global state of interest. Similarly to the case of unconditional breakpoints, a causal chain of *computation* messages beginning at q_ℓ such that $cp_\ell[\ell] =$ **true**, going through a number of q_k's, each with $cp_k[k] =$ **true** as well, and finally reaching q_i with $cp_i[i] =$ **false** requires q_i to take into account what it receives attached to the *computation* message in updating gs_i. On the other hand, if no such q_i is ever reached, then the detected global state has a chance to be an earlier one. These two cases are illustrated in Figure 4, respectively in parts (a) and (b). Maintaining alt_gs_i has the function of allowing this earlier global state to be saved in gs_i, to be used in case no causal chain of the sort we just described ever occurs. The array alt_gs_i is initialized like gs_i and is attached to *computation* messages with its ith component modified to lt_i. A *computation* message arriving at q_i affects alt_gs_i and may eventually affect gs_i, which happens if $cp_i[i] =$ **false** upon arrival of the *computation* message, by simply updating gs_i to alt_gs_i when lp_i becomes **true**. Only in this situation,

or upon the receipt of *broadcast* messages, does gs_i get updated, but then so does alt_gs_i, so $gs_i[k] \leq alt\_gs_i[k]$ for every $p_k \in N$.



(a)



(b)

Figure 4. Earliest global states and breakpoints on stable conjunctive predicates

Actions at q_i for algorithm DETECT_STABLE_CP**:**

(1) Upon detecting that lp_i has become **true**:

$cp_i[i] := lp_i$;
$alt\_gs_i[i] := lt_i$;
for $k := 1$ **to** n **do**
    $gs_i[k] := alt\_gs_i[k]$;
if $cp_i[1] \wedge \cdots \wedge cp_i[n]$ **then**
    $found_i := \mathbf{true}$
else
    Send $(broadcast, cp_i, gs_i, \mathbf{nil})$ to every $q_k \in N_i'$;

(2) Upon receipt of $(broadcast, cp_j, gs_j, \mathbf{nil})$ from $q_j \in N_i'$:

if not $found_i$ **then**
    begin
        $changed_i := \mathbf{false}$;
        for $k := 1$ **to** n **do**
            if $gs_i[k] < gs_j[k]$ **then**
                begin
                    $gs_i[k] := gs_j[k]$;
                    $cp_i[k] := cp_j[k]$;
                    $changed_i := \mathbf{true}$
                end;
        for $k := 1$ **to** n **do**
            if $alt\_gs_i[k] < gs_j[k]$ **then**
                $alt\_gs_i[k] := gs_j[k]$;
        if $cp_i[i]$ **and** $changed_i$ **then**
            if $cp_i[1] \wedge \cdots \wedge cp_i[n]$ **then**
                $found_i := \mathbf{true}$
            else
                Send $(broadcast, cp_i, gs_i, \mathbf{nil})$ to every $q_k \in N_i'$
    end;

(3) Upon receipt of $(body)$ from p_i destined to $p_j \in N_i$:

Forward $\big(computation, cp_i, alt\_gs_i(lt_i), body\big)$ to q_j;

(4) Upon receipt of $(computation, cp_j, gs_j, body)$ from $q_j \in N_i'$:

if not $found_i$ **then**
  for $k := 1$ **to** n **do**
    if $alt\_gs_i[k] < gs_j[k]$ **then**
      begin
        $cp_i[k] := cp_j[k]$;
        $alt\_gs_i[k] := gs_j[k]$
      end;
Forward $(body)$ to p_i;

Correctness and complexity properties of algorithm DETECT_STABLE_CP are established in the following two theorems.

Theorem 9. There exist $p_i \in N$ and $t \geq 0$ such that the following three conditions are equivalent to one another for algorithm DETECT_STABLE_CP.

(a) There exists a global state φ such that $lp_k =$ **true** at time $\varphi[k]$ for all $p_k \in N$;

(b) $found_i$ becomes **true** at time $lt_i = t$;

(c) At time $lt_i = t$, gs_i is the earliest global state at which $lp_k =$ **true** for all $p_k \in N$.

Proof: See Appendix 2. ■

Theorem 10. Algorithm DETECT_STABLE_CP has message complexity of $O\big((c + ne)n \log T\big)$ bits, global time complexity of $O(n)$, local time complexity of $O(n)$ per message reception, and requires $O(n \log T)$ bits of storage per process.

Proof: See Appendix 2. ■

6. Breakpoints on Generic Conjunctive Predicates

In this section we continue our treatment of breakpoints on conjunctive predicates, but no longer assume stability, i.e., every local predicate is now allowed to switch back and forth between being false and true along the computation. This added generality apparently aggravates the problem's difficulties considerably, but an approach quite similar to the one we adopted in Sections 4 and 5 suffices as the basis of our solution. In those sections we were led to the use of the array alt_gs_i by q_i as a means of providing an additional view at q_i of the global state to be detected, so that gs_i could retain the characteristics of an earlier global state to be used when the fully updated alt_gs_i was not needed. In this section too this array is employed (with the same purpose), but the increased complexity of the generic conjunctive case requires that an additional array, called alt_cp_i, be also needed

to accompany alt_gs_i. In contrast with the stable case, this array is needed because local predicates are no longer guaranteed to remain true once they become true.

Furthermore, not only do we need a means of storing potentially earlier global states for use when appropriate, but also we must have a means of coping with the possibility that the local predicates may become true and false several times before becoming true at a local time with which they can participate in a global state. This situation is depicted in Figure 5, in which the partition indicated by a dashed line cannot be a global state.

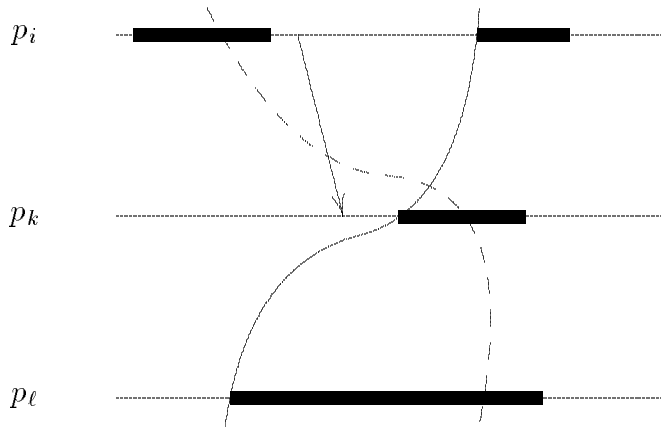


Figure 5. Global states and breakpoints on generic conjunctive predicates

Our approach to the design of algorithm DETECT_CP has been to obtain it as an extension to algorithm DETECT_STABLE_CP to deal with the instability of the local predicates. This extension relies heavily on the assumption, which we now make, that channels in E are FIFO (First In, First Out), i.e., they deliver messages in the order messages are sent. The central issue in obtaining this extension is to ensure that global states at which the conjunctive predicate holds are never missed. Note that this would not be ensured if for $p_i \in N$ we simply added the new array alt_cp_i to algorithm DETECT_STABLE_CP along with a new action to set $alt\_cp_i[i]$ to **false** (and fix $alt\_gs_i[i]$ accordingly) whenever lp_i became **false**. Such a naïve extension would not work even in the absence of *computation* messages, because earliest global states would surely be missed. Consider, for example, part (a) of Figure 6, in which two global states are shown. Of these, the one represented by a solid line is clearly the earliest global state at which the conjunctive predicate holds. Nevertheless, even with the aforementioned addition to algorithm DETECT_STABLE_CP, by action (1) of that algorithm, and depending on the timing of the *broadcast* messages, the global state represented by a dashed line might be the one to be detected.

If *computation* messages were allowed, then the situation would be even worse, because global states at which the conjunctive predicate holds might be missed altogether, thereby making the detection by some processes impossible. This is illustrated in part (b) of Figure 6, in which, as in part (a), the global state represented by a solid line is the earliest global state at which the conjunctive predicate holds. If the *computation* message shown in the figure arrives at q_i before any of the *broadcast* messages originally propagated by q_ℓ, then by actions (4) and (1) (in this order) of (the extended) algorithm DETECT_STABLE_CP, gs_i becomes the global state shown in the figure as a dashed line. What this means is that the detection by q_i of any global state in which the conjunctive predicate holds is made impossible. While in all of our algorithms certain processes never come to actually detect the global state of interest, this is so exclusively because the broadcast does not go beyond a process that does perform the detection. In the case of Figure 6(b), by contrast, q_i would miss all the global states of interest even if reached by a broadcast from a process having already made the detection.

What is needed is to disallow cp_i and alt_cp_i to be updated when q_i is reached by a broadcast originated at $q_k \in N'$ when, respectively, $cp_i[k] = \textbf{true}$ and $alt\_cp_i[k] = \textbf{true}$. This includes the case in which $p_k = p_i$, that is, $cp_i[i]$ and $alt\_cp_i[i]$ are only to be updated if, respectively, $cp_i[i] = \textbf{false}$ and $alt\_cp_i[i] = \textbf{false}$. For this reason, it becomes essential for q_i to know the origin of a broadcast when reached by the corresponding *broadcast* messages, and then we can no longer employ, as we have been doing, algorithm BROADCAST_WHEN_TRUE of Section 3. Our approach to broadcasts will then be to tag their messages with the type $broadcast_k$ for broadcasts originating at q_k.

One might be suspicious, however, that, by precluding cp_i and alt_cp_i from being updated upon receipt of $broadcast_k$ messages (or upon the origination of this broadcast, if $p_k = p_i$) because $cp_i[k] = \textbf{true}$ and $alt\_cp_i[k] = \textbf{true}$, respectively, we might be missing global states at which the conjunctive predicate holds as well. The relevant aspects of this issue are twofold. First of all, such a $broadcast_k$ message must not be conveying any information about another process's, say p_ℓ's for $p_\ell \in N$, predicate that would not be received anyway through a $broadcast_\ell$ message, so long as every broadcast is ensured to reach every node until the breakpoint is detected. Secondly, and this is where the FIFO assumption comes in, if the gs_k accompanying the $broadcast_k$ message were to contribute to the detected global state, then necessarily a causal chain of *computation* messages leaving q_k while $alt\_cp_k[k] = \textbf{false}$ would exist destined to some $q_j \in N'$, where it would arrive when $alt\_cp_j[j] = \textbf{false}$. But then one of two situations would happen involving the gs_k that q_i ignored. If this gs_k arrived at q_j before lp_j became **true**, then would be taken into account by q_j and participate in the broadcast that q_j would generate when lp_j

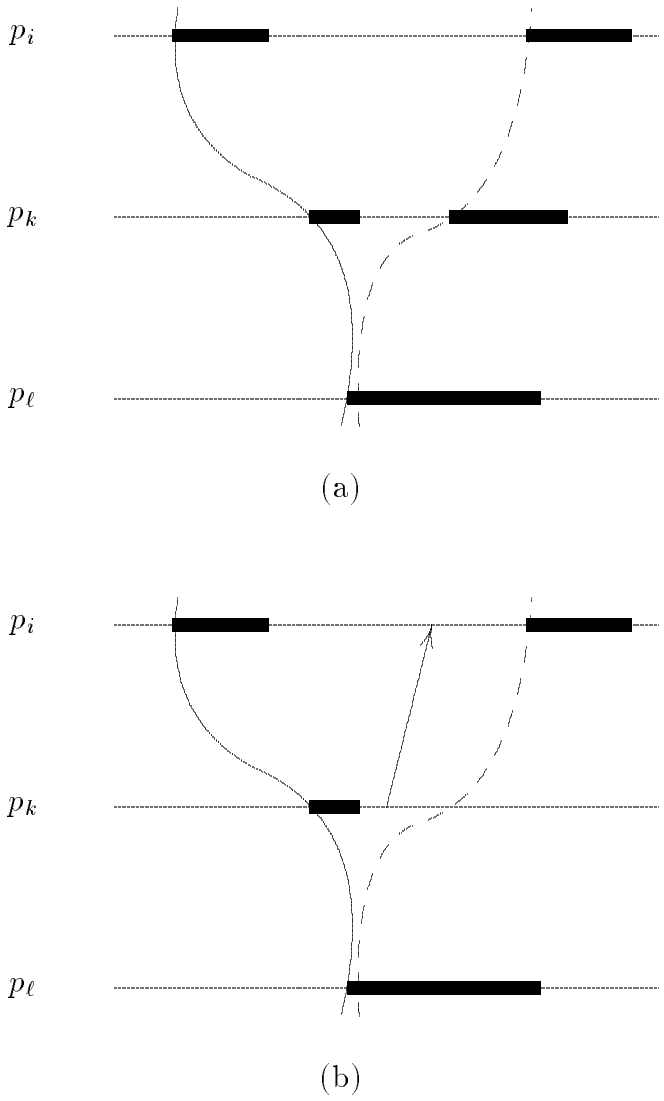


Figure 6. Earliest global states and breakpoints on generic conjunctive predicates

next became **true**. If, on the other hand, the gs_k arrived at q_j after lp_j became **true**, then it would be missed by the broadcast initiated by q_j upon the detection that lp_j had become **true**, but would be taken into account by q_j and participate in the forwarding by q_j of the broadcast initiated by q_k. However, in the latter case, this forwarding by q_j could only be expected to convey the gs_k correctly to all processes if it were treated as a new broadcast initiated at q_j. Then, aside from the already mentioned need to attach a broadcast initiator's identity to broadcast messages, the broadcasts that we need are very similar to the ones we employed earlier in this paper, in the sense that a $broadcast_k$ message arriving at q_i must be forwarded if it causes changes in either $cp_i[k]$ or $alt\_cp_i[k]$.

The variables employed by q_i in algorithm DETECT_CP are those employed in algorithm DETECT_STABLE_CP and the already mentioned array alt_cp_i. They are all initialized as in the previous algorithm, and alt_cp_i is initialized like cp_i, i.e., for $p_k \in N$ their kth components are initially **true** if p_k does not participate in the breakpoint or **false** if p_k does participate. A further assumption regarding variables' values is that lp_i does not ever become **false** if p_i is not one of the processes participating in the breakpoint.

Actions at q_i for algorithm DETECT_CP:

(1) Upon detecting that lp_i has become **true**:

if not $found_i$ **then**
begin
$alt\_cp_i[i] := lp_i$;
$alt\_gs_i[i] := lt_i$;
Send $(broadcast_i, alt\_cp_i, alt\_gs_i, \mathbf{nil})$ to every $q_k \in N_i'$
end;

(2) Upon detecting that lp_i has become **false**:

if not $found_i$ **then**
begin
$alt\_cp_i[i] := lp_i$;
$alt\_gs_i[i] := lt_i$
end;

(3) Upon receipt of $(broadcast_\ell, cp_\ell, gs_\ell, \textbf{nil})$ from $q_j \in N_i'$:

```
if not found_i then
    begin
        changed_i := false;
        if not cp_i[ℓ] then
            for k := 1 to n do
                if gs_i[k] < gs_ℓ[k] then
                    begin
                        gs_i[k] := gs_ℓ[k];
                        cp_i[k] := cp_ℓ[k];
                        changed_i := true
                    end;
        if not alt_cp_i[ℓ] then
            for k := 1 to n do
                if alt_gs_i[k] < gs_ℓ[k] then
                    begin
                        alt_gs_i[k] := gs_ℓ[k];
                        alt_cp_i[k] := cp_ℓ[k];
                        changed_i := true
                    end;
        if changed_i then
            if cp_i[1] ∧ ··· ∧ cp_i[n] then
                found_i := true
            else
                Send (broadcast_ℓ, cp_ℓ, gs_ℓ, nil) to every q_k ∈ N_i'
    end;
```

(4) Upon receipt of $(body)$ from p_i destined to p_j:

Forward $\big(computation, alt\_cp_i, alt\_gs_i(lt_i), body\big)$ to q_j;

(5) Upon receipt of $(computation, cp_j, gs_j, body)$ from $q_j \in N_i'$:

```
if not found_i then
    for k := 1 to n do
        if alt_gs_i[k] < gs_j[k] then
            begin
                alt_gs_i[k] := gs_j[k];
                alt_cp_i[k] := cp_j[k]
            end;
Forward (body) to p_i;
```

Theorems 11 and 12 given next relate to algorithm DETECT_CP's properties. Recall, in Theorem 12, that P stands for the maximum number of times any process's local predicate becomes true.

Theorem 11. There exist $p_i \in N$ and $t \geq 0$ such that the following three conditions are equivalent to one another for algorithm DETECT_CP.

(a) There exists a global state φ such that $lp_k =$ **true** at time $\varphi[k]$ for all $p_k \in N$;

(b) $found_i$ becomes **true** at time $lt_i = t$;

(c) At time $lt_i = t$, gs_i is the earliest global state at which $lp_k =$ **true** for all $p_k \in N$.

Proof: See Appendix 2. ■

Theorem 12. Algorithm DETECT_CP has message complexity of $O\big((c + Pne)n \log T\big)$ bits, global time complexity of $O(n)$, local time complexity of $O(n)$ per message reception, and requires $O(n \log T)$ bits of storage per process.

Proof: See Appendix 2. ■

7. Checkpointing and Rollback Recovery

Checkpointing and rollback recovery are techniques that allow processes to properly continue to execute after the occurrence of failures, without the need to restart from the beginning [17]. Although originally conceived in this context of fault handling, these techniques can also be used in the debugging of message-passing programs, particularly in conjunction with the breakpoint detection algorithms we discussed throughout the paper. All of our algorithms detect global states with certain properties of interest, but after the detection of such a global state by, say, process $q_i \in N'$, information on this detection must be spread through the other processes so that they can halt their counterparts in N for whatever needs to be done that was the purpose of setting the breakpoint in the first place. However, this spreading of a halt order does necessarily reach most nodes when they are in local states further in time than those recorded in the detected global state. This is where checkpointing and rollback-recovery techniques come in, because most processes will, upon receipt of a halt order, be forced to roll back so that the system can be stopped at the desired global state.

Recording checkpoints as the processes run constitutes a research area with its own problems, especially because the issue of whether the assembled sets of locally recorded checkpoints (the *system checkpoints*) constitute global states must be dealt with. Two general approaches to this issue are usually considered. The first approach proceeds to the recording of checkpoints independently at each node, and upon the need to roll the system back to a previous global state the available collection of system checkpoints is checked for those that do constitute global states (e.g., [2]). The second approach is more

conservative, and attempts to ensure that every checkpoint recorded locally is part of a system checkpoint that constitutes a global state. In this case, rolling the system back is a simple matter, as every one of the available system checkpoints is a global state (e.g., [16, 27]).

The technique that as a first approach we propose to use in conjunction with our breakpoint detection algorithms can be thought of as being of the first type we just described, although fortunately it does not suffer from the possibility that the resulting system checkpoints may not constitute global states. The general technique is rather simple, and takes advantage of the fact that every component in the gs_i that $q_i \in N'$ detects falls into one of the following categories. Either the component is equal to zero, or the component is the local time at which a process had the local condition with which it participates in the breakpoint satisfied, or yet the component is the local time at which a *computation* message was sent. So if every process records a checkpoint at the beginning of its computation, another one whenever its local condition gets satisfied, and another whenever it sends a *computation* message, then once gs_i is detected by q_i every $q_k \in N'$ has simply to roll p_k back to local time $gs_i[k]$.

Of course this strategy places storage requirements in addition to those given by Theorems 4, 8, 10 and 12, but often improvements can be made to the overall strategy to minimize the need for storage at each process. For example, for the detection of a breakpoint on a disjunctive predicate, q_i records checkpoints only until lp_i becomes **true** (if p_i is indeed participating in the breakpoint) or until a *computation* message carrying $bit_j =$ **true** is received from $q_j \in N_i'$ such that (regardless of whether p_i participates in the breakpoint). Upon either occurrence, only the checkpoint whose recording time is the updated $gs_i[i]$ needs to be retained, while the others may be done away with. It should be noted that processes that do not participate in the breakpoint may have to record checkpoints further on until a halt order is received.

For the detection of an unconditional breakpoint, a process $p_i \in N$ participating in the breakpoint needs a single checkpoint at time lub_i. If p_i does not participate in the breakpoint, then q_i, upon receiving a *broadcast* message carrying gs_j from $q_j \in N_i'$ such that, needs no longer record checkpoints upon sending *computation* messages to q_j if $ub_j[j] =$ **true**, and may in addition discard every checkpoint recorded earlier than $gs_j[i]$.

For the detection of breakpoints on conjunctive predicates, a process $q_i \in N'$ records checkpoints when lp_i becomes **true** and upon sending *computation* messages, but only while $lp_i =$ **true** (if p_i does participate in the breakpoint) or at all times (if p_i does not participate in the breakpoint, being therefore regarded as if its "predicate" were always true). If the conjunctive predicate is stable, then the receipt of a *broadcast* message carrying gs_j from

$q_j \in N_i'$ allows q_i to discard all the checkpoints recorded earlier than $gs_j[i]$. This is similar to the case of unconditional breakpoints, but it should be noted that q_i must proceed with the recording of checkpoints upon sending *computation* messages to q_j because, in contrast with the former case, the local time with which p_j participates in the desired global state may not yet have occurred (even though it must already have reached a situation in which its local predicate is true). If the conjunctive predicate is not stable, then all that may be done upon receipt of the *broadcast* message is to discard every checkpoint recorded earlier than the updated $gs_i[i]$, as they will definitely not take part in the global state at which the predicate holds.

The overall strategy we have presented so far needs to be improved so that the global checkpoints include messages in transit as well. This measure, although inessential to the rollback process, is essential to restart the execution after the system has been examined at the breakpoint. Possible approaches have been discussed elsewhere [26, 30].

8. Concluding Remarks

In this paper we have considered the problem of designing distributed algorithms for the detection of breakpoints in message-passing programs. Along with the ability to deterministically re-execute such programs, the setting of breakpoints where the program's context can be analyzed stands as a fundamental cornerstone of message-passing program debugging.

We have introduced and analyzed (for both correctness and complexity) four breakpoint-detection algorithms, specifically one for the detection of unconditional breakpoints (algorithm DETECT_UBP) and three for the detection of conditional breakpoints. Of the latter, one is for the detection of breakpoints on disjunctive predicates (algorithm DETECT_DP), one for the detection of breakpoints on stable conjunctive predicates (algorithm DETECT_STABLE_CP), and finally one for the detection of breakpoints on generic conjunctive predicates (algorithm DETECT_CP). To our knowledge, these are the first fully distributed algorithms for the type of breakpoint detection they perform, in the sense that no centralized entity needs to be singled out to handle the task. We have, nevertheless, added a brief discussion of other related algorithms, although we refrained from attempting any deeper comparison (e.g., complexity-based), as the algorithms we presented and those available in the literature are inherently different.

Except for algorithm DETECT_DP, whose global time complexity is of $O(1)$, all algorithms have $O(n)$ global time complexity. All four algorithms have the same local time complexity of $O(n)$ per message reception. The storage requirement of all the algorithms is of $O(n \log T)$ bits per process, where T is an upper bound on the local times

at all processes. If the computation proper employs $O(c)$ messages, then algorithm DETECT_DP has a message complexity of $O(cn \log T)$ bits, while algorithms DETECT_UBP and DETECT_STABLE_CP have the same message complexity of $O\big((c + ne)n \log T\big)$ bits. Algorithm DETECT_CP has a message complexity of $O\big((c + Pne)n \log T\big)$ bits, where P is the maximum number of times any local predicate becomes true.

All the breakpoints our algorithms detect are earliest global states with respect to the particular property involved. In order to actually bring the computation to a halt at that exact global state, our suggestion has been to employ checkpointing and rollback-recovery techniques. Clearly, additional memory requirements are involved then, and we have pointed out a first approach to specific strategies for handling the checkpointing efficiently in each of the four cases.

Acknowledgements

The authors have received partial support from the Brazilian agencies CAPES and CNPq.

References

1. Becher, J. D., and McDowell, C. E. Debugging the MP-2001. *Proc. of New Frontiers, A Workshop on Future Directions of Massively Parallel Processing*, 1992, pp. 48–57.

2. Bhargava, B., and Lian, S. R. Independent checkpointing and concurrent rollback for recovery — An optimistic approach. *Proc. of the IEEE Symposium on Reliable Distributed System*, 1988, pp. 3–12.

3. Chandy, K. M., and Lamport, L. Distributed snapshots: determining global states of distributed systems. *ACM Trans. on Computer Systems* **3** (1985), 63–75.

4. Choi, J., Miller, B. P., and Netzer, R. H. B. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. on Programming Languages and Systems* **13** (1991), 491–530.

5. Cooper, R., and Marzullo, K. Consistent detection of global predicates. *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, 1991, pp. 167–174.

6. Drummond, L. M. de A., and Barbosa, V. C. Distributed breakpoint detection in message-passing programs. Tech. Rep. ES-306/94, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, Brazil, July 1994.

7. Fidge, C. Logical time in distributed computing systems. *IEEE Computer* **24** (1991), 28–33.

8. Garcia-Molina, H., Germano, F., and Kohler, W. Debugging a distributed computing system. *IEEE Trans. on Software Engineering* **10** (1984), 210–219.

9. Garg, V. K., and Tomlinson, A. I. Detecting relational global predicates in distributed systems. *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993, pp. 21–31.

10. Garg, V. K., and Waldecker, B. Detection of unstable predicates in distributed programs. *Proc. of the Twelfth Conference on Foundations of Software Technology & Theoretical Computer Science*, 1992, pp. 253–264.

11. Garg, V. K., and Waldecker, B. Detection of weak and unstable predicates in distributed programs. *IEEE Trans. on Parallel and Distributed Systems* **5** (1994), 299–307.

12. Goldberg, A. P., Gopal, A., Lowry, A., and Strom, R. Restoring consistent global states of distributed computations. *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, 1991, pp. 144–154.

13. Haban, D., and Weigel, W. Global events and global breakpoints in distributed systems. *Proc. of the Twenty-First International Conference on System Sciences, Vol. 2*, 1988, pp. 166–175.

14. Hurfin, M., Plouzeau, N., and Raynal, M. Detecting atomic sequences of predicates in distributed computations. *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1993, pp. 32–40.

15. Joyce, J., Lomow, G., Slind, K., and Unger, B. W. Monitoring distributed systems. *ACM Trans. on Computer Systems* **5** (1987), 121–150.

16. Kim, K. H., You, J. H., and Abouelnaga, A. A scheme for coordinated execution of independently designed recoverable distributed processes. *Proc. of the IEEE Fault-Tolerant Computing Symposium*, 1986, pp. 130–135.

17. Koo, R., and Toueg, S. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. on Software engineering* **13** (1987), 23–31.

18. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM* **21** (1978), 558–565.

19. Lamport, L., and Lynch, N. Distributed computing: models and methods. In van Leeuwen, J. (Ed.). *Handbook of Theoretical Computer Science, Vol. B.* The MIT Press, Cambridge, MA, 1990, pp. 1156–1199.

20. LeBlanc, T., and Mellor-Crummey, J. M. Debugging parallel programs with instant replay. *IEEE Trans. on Computers* **36** (1987), 471–482.

21. Manabe, Y., and Imase, M. Global conditions in debugging distributed programs. *J. of Parallel and Distributed Computing* **15** (1992), 62–69.

22. Mattern, F. Virtual time and global states of distributed systems. *Proc. of the 1988 International Workshop on Parallel and Distributed Algorithms*, 1989, pp. 215–226.

23. McDowell, C., and Helmbold, D. Debugging concurrent programs. *ACM Computing Surveys* **21** (1989), 593–622.

24. Miller, B., and Choi, J. Breakpoints and halting in distributed programs. *Proc. of the Eighth International Conference on Distributed Computing Systems*, 1988, pp. 316–323.

25. Miller, B., and Choi, J. A mechanism for efficient debugging of parallel programs. *Proc. of the Sigplan'88 Conference on Programming Language Design and Implementation*, 1988, pp. 135–144.

26. Netzer, R. H. B., and Xu, J. Adaptive message logging for incremental program replay. *IEEE Parallel & Distributed Technology* **1** (1993), 32–39.

27. Ramanathan, P., and Shin, K. G. Use of common time base for checkpointing and rollback recovery in a distributed system. *IEEE Trans. on Software Engineering* **19** (1993), 571–583.

28. Spezialetti, M. An approach to reducing delays in recognizing distributed event occurrences. *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, 1991, pp. 155–166.

29. Spezialetti, M., and Kearns, J. P. Simultaneous regions: a framework for the consistent monitoring of distributed systems. *Proc. of the Ninth International Conference on Distributed Computing Systems*, 1989, pp. 61–68.

30. Xu, J., and Netzer, R. H. B. Adaptive independent checkpointing for reducing rollback propagation. *Proc. of the Fifth IEEE Symposium on Parallel and Distributed Processing*, 1993, pp. 754–761.

Appendix 1. Summary of Notation

In this appendix we provide a brief summary of the notation used throughout the paper. Included in this summary is only the notation defined relatively early in the paper, but whose usage spans most of the sections.

- N: Set of processes of the computation proper;
- E: Set of bidirectional communication channels;
- n: The cardinality of N;
- e: The cardinality of E;
- N_i: Set of neighbors of $p_i \in N$;
- N': Set of processes that perform the breakpoint detection;
- N'_i: Set of processes $q_j \in N'$ such that $p_j \in N_i$;
- lub_i: Local time giving the unconditional breakpoint for $p_i \in N$;
- lp_i: Local predicate for $p_i \in N$;
- lt_i: Local time for $p_i \in N$;
- $c(n, e)$: Message complexity (in number of messages) of the computation proper;
- T: Maximum local time for any process in N;
- P: Maximum number of times a local predicate becomes true for any process in N.

Appendix 2. Lemma and Theorem Proofs

This appendix is dedicated to the presentation of the proofs of Lemmas 1 and 2 and of Theorems 3 through 12. Of these, the correctness-related theorems state the equivalence among several conditions. The proof strategy is then in all these cases to show that the first condition implies the second, which implies the third, and so on, and finally to show that the last condition implies the first.

Proof of Lemma 1

If $gs_i(lt_i)$ is not a global state, then there must exist $p_k, p_\ell \in N$ such that a *computation* message was sent by p_k strictly later than $gs_i(lt_i)[k]$ and received at p_ℓ earlier than (or at) $gs_i(lt_i)[\ell]$. By the definition of $gs_i(lt_i)$, and by hypothesis, it follows that the message must have been sent later than $gs_i[k]$ and arrived at p_ℓ earlier than (or at) $gs_i[\ell]$, and then gs_i must not be a global state, which is a contradiction. ■

Proof of Lemma 2

Let φ'' be the component-wise maximum of φ and φ', and suppose that it is not a global state. Then there must exist $p_k, p_\ell \in N$ such that a message was sent by p_k strictly later than $\varphi''[k]$ and received at p_ℓ earlier than (or at) $\varphi''[\ell]$. Because $\varphi''[k] \geq \varphi[k]$ and $\varphi''[k] \geq \varphi'[k]$, then φ must not be a global state if $\varphi''[\ell] = \varphi[\ell]$. Likewise, if $\varphi''[\ell] = \varphi'[\ell]$, then φ' must not be a global state. Either case yields a contradiction. ■

Proof of Theorem 3

(a) ⇒ (b):

At least one of the processes $p_k \in N$ for which lp_k ever becomes **true** must by actions (2) and (3) have reached this state for the first time when $found\_elsewhere_k$ = **false**. The assertion then follows immediately by action (1), with p_i being this particular process and t being the local time at which lp_i becomes **true** for the first time.

(b) ⇒ (c):

By hypothesis and by action (1), $found\_elsewhere_i$ can only have become **true** after time t. By Lemmas 1 and 2, the gs_i produced by action (1), the $gs_i(lt_i)$ used in action (2), and the gs_i yielded by action (3) must all be global states. As a consequence of this, by action (1) gs_i is at time t a global state at which lp_i = **true**. If gs_i were not an earliest global state at which lp_k = **true** for at least one $p_k \in N$, then either $found\_elsewhere_i$ would by actions (2) and (3) have become **true** prior to t, and then $found_i$ would be **false**

at t, which is a contradiction, or lp_k would for some $p_k \in N$ be **true** right from the start, which is ruled out by our assumption on the initial values of these variables.

(c) $\Rightarrow$ (a):

This is immediate. ■

Proof of Theorem 4

Each of the $O(c)$ *computation* messages carries an n-component array, each of whose components is an integer no larger than T, thence the message complexity of $O(cn \log T)$ bits. Because only *computation* messages are employed, the global time complexity is of $O(1)$. Each message reception requires $O(n)$ comparisons, thence the local time complexity. For $p_i \in N$, process q_i needs to store the array gs_i, which requires $O(n \log T)$ bits. ■

Proof of Theorem 5

(a) $\Rightarrow$ (b):

If exactly one process participates in the global condition, then by action (1) $found_i$ becomes **true**, with $p_i \in N$ being this process and t the time at which lc_i becomes **true**. No messages are ever sent in this case. If at least two processes participate, then at least one of them, say $p_k \in N$, is such that q_k does by action (1) send a *broadcast* message to its neighbors when lc_k becomes **true**, which by action (2) pass the updated information on, so long as the update introduced changes and their local conditions hold as well. Because this broadcast carries lc_k, it must introduce changes when reaching every node for the first time and is therefore propagated. This happens to the local condition of every participating node, and then at least one process, say q_i, upon having been reached by their broadcasts, and having $lc_i =$ **true**, sets $found_i =$ **true**. The value of t here is either the time at which the last broadcast to reach q_i does reach it by action (2) or the time at which lc_i becomes **true** by action (1).

(b) $\Rightarrow$ (c):

By Lemmas 1 and 2, the gs_i produced in actions (1) and (2) are global states. Consequently, and by actions (1) and (2) as well, at time t gs_i is a global state at which $lc_k =$ **true** for all $p_k \in N$. That gs_i is the earliest such global state is immediate, because of the absence of *computation* messages, which implies that $gs_i[k]$ is either zero or the time at which lc_k becomes **true**.

(c) $\Rightarrow$ (a):

This is immediate. ■

Proof of Theorem 6

The worst case is that in which all nodes start the algorithm concurrently, and furthermore the broadcast started by a node traverses all channels. Because two n-component arrays are sent along with each message, one comprising single-bit components, the other integers bounded by T, the message complexity becomes $O(n^2 e \log T)$ bits. No causal chain of messages comprises more than $O(n)$ messages, because this is what it takes for a broadcast to reach all nodes, thence the global time complexity. The local time complexity and the storage requirement are like those of algorithm DETECT_DP, therefore given by Theorem 4. ■

Proof of Theorem 7

(a) $\Rightarrow$ (b):

Suppose that there does exist $p_k \in N$ such that in_error_k becomes **true**. By action (4), this must happen upon receipt, when $ub_k[k] =$ **false**, of a *computation* message contained in a causal chain of *computation* messages started at, say, $q_\ell \in N'$, sent when $ub_\ell[\ell] =$ **true**. No array φ such that $\varphi[k] = lub_k$ and $\varphi[\ell] = lub_\ell$ can then be a global state, and because both $lub_k < \infty$ and $lub_\ell < \infty$, we have a contradiction.

(b) $\Rightarrow$ (c):

If in_error_k never becomes **true** for any $p_k \in N$, then actions (1) and (2) are, so far as *broadcast* messages are concerned, identical to actions (1) and (2), respectively, of algorithm BROADCAST_WHEN_TRUE. This part of the proof is then analogous to the (a) $\Rightarrow$ (b) part in the proof of Theorem 5.

(c) $\Rightarrow$ (d):

By Lemmas 1 and 2, the gs_i produced by action (1), the $gs_i(lt_i)$ and $alt\_gs_i(lt_i)$ used in action (3), and the gs_i and alt_gs_i produced by actions (2) and (4) are all global states. This implies, by actions (1) and (2) and at time t, that gs_i is a global state at which $ub_i[k] \neq$ **false** for all $p_k \in N$, or, equivalently, a global state such that $gs_i[k] = lub_k$ for every $p_k \in N$ such that $lub_k < \infty$. In order to show that gs_i is the earliest global state with these characteristics, consider any other n-component array of local times, call it φ, such that $\varphi[k] = gs_i[k]$ for all $p_k \in N$ such that $lub_k < \infty$, and $\varphi[k] < gs_i[k]$ for at least one $p_k \in N$ such that $lub_k = \infty$. For this particular p_k, in order for $gs_i[k]$ to have been assigned the value greater than $\varphi[k]$, a causal chain of *computation* messages must

have existed from p_k (leaving at time $gs_i[k]$) to some $p_\ell \in N$, where by action (4) it must have arrived at q_ℓ when $ub_\ell[\ell] = \textbf{false}$ (otherwise gs_ℓ would not have been updated, and so neither would gs_i through the broadcast). In addition, because in_error_ℓ must have remained **false**, every process involved in this chain (except for q_ℓ but including q_k) must have had an **undefined** in its local record of its local unconditional breakpoint (for q_k, $ub_k[k] = \textbf{undefined}$). But because $ub_\ell[\ell]$ was found to be **false**, φ cannot possibly be a global state such that $\varphi_k = lub_k$ for all $p_k \in N$ such that $lub_k < \infty$.

(d) $\Rightarrow$ (a):

This is immediate. ■

Proof of Theorem 8

The message complexity of this algorithm is the sum of the message complexities of algorithm DETECT_DP and algorithm BROADCAST_WHEN_TRUE. By Theorems 4 and 6, we obtain $O\big((c + ne)n \log T\big)$ bits. The remaining complexities and storage requirement are exactly those of algorithm BROADCAST_WHEN_TRUE, and are then given as in Theorem 6. ■

Proof of Theorem 9

(a) $\Rightarrow$ (b):

Actions (1) and (2) are, from the standpoint of *broadcast* messages alone, identical to actions (1) and (2), respectively, of algorithm BROADCAST_WHEN_TRUE. This part of the proof then goes along the same lines as the (a) $\Rightarrow$ (b) part in the proof of Theorem 5, so long as no *computation* message overruns any *broadcast* message on any channel. When this happens, however, propagation of the *broadcast* message may by action (2) be interrupted after traversing the channel, specifically upon arriving, say at $q_k \in N'$, and by action (2) finding $cp_k[k] = \textbf{true}$ without causing changes to gs_k or to cp_k. This is so because the gs_j carried by the *broadcast* message is no greater than gs_k in any component, which in turn was updated by action (1) when lp_k became **true** with the alt_gs_k produced by action (4) upon receipt of the *computation* message. The broadcast that by action (1) q_k then initiates when lp_k becomes **true** allows the proof to proceed like that of the (a) $\Rightarrow$ (b) part in the proof of Theorem 5 as well.

(b) $\Rightarrow$ (c):

By Lemmas 1 and 2, the gs_i and alt_gs_i produced by actions (1) and (2), the $alt\_gs_i(lt_i)$ used in action (3), and the alt_gs_i produced by action (4) must all be global

states. A consequence of this is that, by actions (1) and (2), gs_i is at time t a global state at which $cp_i[k] = \textbf{true}$ for all $p_k \in N$. To show that gs_i is the earliest such global state requires that we consider any other n-component array of local times, call it φ, such that $lp_k = \textbf{true}$ at time $\varphi[k]$ for all $p_k \in N$ and such that $\varphi[k] < gs_i[k]$ for at least one $p_k \in N$. For this particular p_k, $gs_i[k]$ can only have been assigned the value greater than $\varphi[k]$ if a causal chain of *computation* messages existed from p_k (leaving at time $gs_i[k]$) to some $p_\ell \in N$, which by action (1) must have arrived at q_ℓ when $cp_\ell[\ell] = \textbf{false}$ (otherwise gs_ℓ would not have been updated, and so neither would gs_i by means of the broadcast). But because $cp_\ell[\ell]$ was found to be **false**, φ cannot possibly be a global state such that $lp_k = \textbf{true}$ at time $\varphi[k]$ for all $p_k \in N$.

(c) $\Rightarrow$ (a):

This is immediate. ■

Proof of Theorem 10

The complexities and storage requirement for this algorithm are the same as those of algorithm DETECT_UBP, therefore given as in Theorem 8. ■

Proof of Theorem 11

(a) $\Rightarrow$ (b):

Let φ' be the earliest global state such that $lp_k = \textbf{true}$ for all $p_k \in N$. If exactly one process participates in the breakpoint, then by actions (1) and (3) the assertion follows trivially, with p_i being any of that process's neighbors and t the time at which the broadcast reaches it. If more than one process participates in the breakpoint, then there are two major cases to be considered.

In the first major case, at φ', and for every participating process $p_k \in N$, lp_k has become **true** exactly once. If lp_k never becomes **true** again for any participating $p_k \in N$, then actions (1) and (3) ensure that at least one process $p_i \in N$ is reached by all broadcasts and sets $found_i$ to **true** upon being reached, at time t, by the last broadcast. If, on the other hand, at least one of the participating processes, say $p_k \in N$, is such that lp_k becomes **true** at least twice, then no $p_i \in N$ has $cp_i[k] = \textbf{false}$ upon being reached by the broadcasts other than the first initiated by q_k, and then by actions (1) and (3), as previously, at least one process $p_i \in N$ sets $found_i$ to **true** at time t. To see why $cp_i[k]$ must be **true** when (and if) q_i is reached by the broadcasts other than the first initiated by q_k, suppose for a moment that $cp_i[k]$ were found **false** when q_i was reached by any of

those broadcasts. By actions (1), (2), (3), and (5), $cp_i[k]$ can only have become **false** upon receipt of a broadcast message, whose sending must have been influenced by the receipt of a causal chain of *computation* messages starting at p_k carrying the **false** value. Because channels are FIFO, this chain must have left p_k after q_k initiated its first broadcast, or the broadcast that set $cp_i[k]$ to **false** would not have prevailed. But if this broadcast was started at, say, $q_\ell \in N'$, then q_i must have had $cp_i[\ell] =$ **false** upon being reached by it, and then either this was the first broadcast started by q_ℓ or the entire argument must be repeated with p_ℓ in place of p_k. Eventually, in this argument we would have to end up in a causal chain of *computation* messages (influencing one of those broadcasts) which was started at a process after φ' and terminated at another process earlier than φ', and then φ' would not be a global state.

In the second major case, at φ' at least one of the participating processes, say $p_k \in N$, is such that lp_k has become **true** at least twice. In this case, in each of the time intervals during which $lp_k =$ **false** preceding φ' a causal chain of *computation* messages exists leaving p_k and arriving at another participating process, say $p_\ell \in N$, such that lp_ℓ becomes **true** exactly once prior to φ'. At q_ℓ, this chain has the effect of causing $cp_\ell[k]$ to be set to **false**, so that the broadcast initiated by q_k when lp_k becomes **true** to remain **true** through φ' does by action (3) cause changes in cp_ℓ and is therefore propagated. This may happen either before lp_ℓ becomes **true**, and then the broadcast initiated by q_ℓ when lp_ℓ becomes **true** carries along with it the information on lp_k's becoming **true**, or it may happen after lp_ℓ becomes **true**, in which case the $broadcast_k$ messages that q_ℓ forwards follow the $broadcast_\ell$ messages sent previously, possibly setting to **true** whichever $cp_i[k] =$ **false** it may find for some $p_i \in N$. The remaining of the argument proceeds like in the first major case.

(b) $\Rightarrow$ (c):

By Lemmas 1 and 2, the alt_gs_i produced by actions (1) and (2), the gs_i and alt_gs_i obtained in action (3), the $alt\_gs_i(lt_i)$ used in action (4), and the alt_gs_i obtained in action (5) are all global states. Consequently, by action (3), at time t gs_i is a global state at which $lp_k =$ **true** for all $p_k \in N$. We show that gs_i is the earliest global state with these characteristics by considering any other n-component array of local times, call it φ, such that $lp_k =$ **true** at time $\varphi[k]$ for all $p_k \in N$ and furthermore $\varphi[k] < gs_i[k]$ for at least one $p_k \in N$. For this particular p_k, $gs_i[k]$ can only have acquired the value greater than $\varphi[k]$ in one of two cases.

In the first case, a causal chain of *computation* messages must have existed from p_k to some $p_\ell \in N$, which must have left p_k at time $gs_i[k]$ when at q_k $alt\_cp_k[k]$ was **true** and must have arrived at q_ℓ when $cp_\ell[\ell] =$ **false** earlier than $\varphi[\ell]$, and consequently φ cannot

be a global state at which $lp_k = \mathbf{true}$ for all $p_k \in N$.

In the second case, the causal chain of *computation* messages leaving p_k was started when $alt\_cp_k[k] = \mathbf{false}$ at q_k, although $cp_\ell[\ell]$ was as in the first case (in particular, $cp_\ell[\ell] = \mathbf{false}$ earlier than $\varphi[\ell]$). By action (5), and by the FIFO assumption, $alt\_cp_\ell[k]$ must have become **false** and remained so until a new broadcast was started by q_k at time $gs_i[k]$, when $alt\_cp_k[k]$ became **true**. This broadcast may have reached q_ℓ either before or after lp_ℓ became **true**. In the former case, the broadcast that q_ℓ then started carried $alt\_cp_\ell[k] = \mathbf{true}$, which upon reaching q_i and finding $cp_i[\ell] = \mathbf{false}$, by action (3) set $cp_i[k] = \mathbf{true}$. In the latter case, the broadcast that q_ℓ initiated carried $alt\_cp_\ell[k] = \mathbf{false}$ and caused $cp_i[k]$ to be set to **false**. But the forwarding that q_ℓ performed upon receiving the broadcast originated by q_k, which updated $alt\_cp_\ell[k]$ to **true**, carried this value on to q_i, and then $cp_i[k]$ was set to **true**. But then as in the first case φ cannot be a global state at which $lp_k = \mathbf{true}$ for all $p_k \in N$.

(c) $\Rightarrow$ (a):

This is immediate. ■

Proof of Theorem 12

Every node may start as many as P broadcasts, each of which is forwarded by every other node no more than twice, by action (3) and considering that no message from a given broadcast arriving at q_i alters either gs_i or alt_gs_i (equivalently, cp_i or alt_cp_i) more than once. The message complexity of $O\big((c + Pne)n \log T\big)$ bits then follows from Theorem 4 and from the fact that in the worst case each broadcast traverses every channel twice in each direction. The remaining measures are given directly by Theorem 10. ■