

# Detecting Locally Stable Predicates without Modifying Application Messages

Ranganath Atreya<sup>1</sup>, Neeraj Mittal<sup>1</sup>, and Vijay K. Garg<sup>2\*</sup>

<sup>1</sup> Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA

atreya@student.utdallas.edu      neerajm@utdallas.edu

<sup>2</sup> Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, TX 78712, USA  
garg@ece.utexas.edu

**Abstract.** In this paper, we give an efficient algorithm to determine whether a locally stable predicate has become true in an underlying computation. Examples of locally stable predicates include termination and deadlock. Our algorithm does not require application messages to be modified to carry control information (*e.g.*, vector timestamps), nor does it inhibit events (or actions) of the underlying computation. Once the predicate becomes true, the detection latency (or delay) of our algorithm is proportional to the time-complexity of computing a (possibly inconsistent) snapshot of the system. Moreover, only  $O(n)$  control messages are required to detect the predicate once it holds, where  $n$  is the number of processes.

## 1 Introduction

Two important problems in distributed systems are detecting termination of a distributed computation and detecting deadlock in a distributed database system. Termination and deadlock are examples of *stable properties*. A property is stable if it never becomes false once it becomes true. For example, once a subset of processes are involved in a deadlock, they continue to stay in a deadlocked state. An algorithm to detect a general stable property involves collecting the relevant states of processes and channels that are consistent with each other and testing to determine whether the property holds over the collected state. By repeatedly taking such *consistent snapshots* of the computation and evaluating the property over the collected state, it is possible to eventually detect a stable property once it becomes true.

Several algorithms have been proposed in the literature for computing a consistent snapshot of a computation [1–5]. These algorithms can be broadly classified into four categories. They either require sending a control message along every channel in the system [1] or rely on piggybacking control information

---

\* Supported in part by the NSF Grants ECS-9907213, CCR-9988225, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant.

on application messages [2] or assume that messages are delivered in causal order [4, 5] or are inhibitory in nature [3]. As a result, consistent snapshots of a computation are expensive to compute. More efficient algorithms have been developed for termination and deadlock that do not require taking consistent snapshots of the computation (*e.g.*, [6–16]).

Termination and deadlock are examples of stable properties that can be formulated as *locally stable predicates* [17]. A predicate is *locally stable* if no process involved in the predicate can change its state relative to the predicate once the predicate holds. In this paper, we show that it is possible to detect *any* locally stable predicate by repeatedly taking possibly inconsistent snapshots of the computation in a certain manner. Since snapshots are not required to be consistent, it is not necessary to send a control message along every channel of the system. Our algorithm does not inhibit any event of the underlying computation nor does it require channels to be FIFO. Unlike Marzullo and Sabel’s algorithm for detecting a locally stable predicate [17], no control information is required to be piggybacked on application messages and therefore application messages do not need to be modified at all. Once the predicate becomes true, the detection latency (or delay) of our algorithm is proportional to the time-complexity of the fastest snapshot protocol. Furthermore, since our approach does not require snapshots to be consistent, it is not necessary to send a control message along every channel of the system when a snapshot is taken.

Our algorithm also unifies several known algorithms for detecting termination and deadlock [6, 8–11]. Some of the examples include Safra’s color-based algorithm [9], Mattern’s four counter algorithm [8] and Mattern *et al*’s sticky-flag algorithm [10] for termination detection, and Ho and Ramamoorthy’s two-phase algorithm [6] for deadlock detection. All of these algorithms can be derived as special cases of the algorithm given in this paper. Note that the two-phase deadlock detection algorithm as described in [6] is actually flawed but can be easily fixed using the ideas given in this paper. Therefore this paper presents a unifying framework for understanding and describing various termination and deadlock detection algorithms.

Although our algorithm does not require application messages to be modified to assist the detection algorithm, it does assume the ability to monitor changes in the values of the relevant variables. This may require modification of the application program. Most of the algorithms for predicate detection make the same assumption and, therefore, may require the application program to be modified to aid in the detection process (*e.g.*, [6, 8, 10–12, 16]).

The paper is organized as follows. Section 2 describes the system model and the notation used in this paper. An algorithm for detecting a locally stable predicate is discussed in Section 3. Due to the lack of space, proofs of lemmas and theorems have been omitted. In Section 4, we analyze the performance of the algorithm. We discuss the related work in Section 5. Finally, Section 6 concludes the paper and also outlines directions for future research.

## 2 Model and Notation

### 2.1 Distributed Computations

We assume an asynchronous distributed system comprising of multiple processes which communicate with each other by sending messages over a set of channels. There is no common clock or shared memory. Processes are non-faulty and channels are reliable. Channels may be non-FIFO. Message delays are finite but may be unbounded.

Processes execute *events* and change their states. A *local state* of a process, therefore, is given by the sequence of events it has executed so far starting from the *initial state*. Events are either *internal* or *external*. An external event could be a *send event* or a *receive event* or both. An event causes the local state of a process to be updated. In addition, a send event causes a message or a set of messages to be sent and a receive event causes a message or a set of messages to be received. The event executed immediately before  $e$  on the same process (as  $e$ ) is called the *predecessor event* of  $e$  and is denoted by  $pred(e)$ . The *successor event* of  $e$ , denoted by  $succ(e)$ , can be defined in a similar fashion.

Although it is possible to determine the exact order in which events were executed on a single process, it is, in general, not possible to do so for events executed on different processes. As a result, an execution of a distributed system, referred to as *distributed computation* (or simply a *computation*), is modeled by an (irreflexive) partial order on a set of events. The partial order, denoted by  $\rightarrow$ , is given by the Lamport's *happened-before relation* (also known as *causality relation*) [18] which is defined as the smallest transitive relation satisfying the following properties:

1. if events  $e$  and  $f$  occur on the same process, and  $e$  occurred before  $f$  in real time then  $e$  happened-before  $f$ , and
2. if events  $e$  and  $f$  correspond to the send and receive, respectively, of a message then  $e$  happened-before  $f$ .

Intuitively, the Lamport's happened-before relation captures the maximum amount of information that can be deduced about ordering of events when the system is characterized by unpredictable message delays and unbounded relative processor speeds.

### 2.2 Cuts, Consistent Cuts and Frontiers

A state of a distributed system, referred to as *global state*, is the collective state of processes and channels. (A channel state is given by the set of messages in transit.) If every process maintains a log of all the messages it has sent and received so far, then a channel state can be determined by examining the state of the two processes connected by the channel. Therefore, in this paper, we view a global state as a collection of local states. The equivalent notion based on events is called *cut*. A cut is a collection of events closed under predecessor relation. In

other words, a cut is a set of events such that if an event is in the set, then its predecessor, if it exists, also belongs to the set. Formally,

$$C \text{ is a cut} \quad \triangleq \quad \langle \forall e, f :: (e = \text{pred}(f)) \wedge (f \in C) \Rightarrow e \in C \rangle$$

The *frontier* of a cut consists of those events of the cut whose successors do not belong to the cut. Formally,

$$\text{frontier}(C) \quad \triangleq \quad \{ e \in C \mid \text{succ}(e) \text{ exists} \Rightarrow \text{succ}(e) \notin C \}$$

Not every cut corresponds to a valid state of the system. A cut is said to be *consistent* if it contains an event only if it also contains all events that happened-before it. Formally,

$$C \text{ is a consistent cut} \quad \triangleq \quad \langle \forall e, f :: (e \rightarrow f) \wedge (f \in C) \Rightarrow e \in C \rangle$$

Observe that if a cut is not consistent then it contains an event such that one or more events that happened-before it do not belong to the cut. Such a scenario, clearly, cannot occur in a real world. Consequently, if a cut is not consistent then it is not possible for the system to be in a global state given by the cut. In other words, only those cuts which are consistent can possibly occur during an execution.

### 2.3 Global Predicates

A *global predicate* (or simply a *predicate*) is defined as a boolean-valued function on variables of one or more processes. In other words, a predicate maps every consistent cut of a computation to either **true** or **false**. Given a consistent cut, a predicate is evaluated with respect to the values of the relevant variables in the state resulting after executing all events in the cut. If a predicate  $b$  evaluates to true for a cut  $C$ , we say that  $C$  *satisfies*  $b$  or, equivalently,  $b(C) = \text{true}$ . Hereafter, we abbreviate expressions  $b(C) = \text{true}$  and  $b(C) = \text{false}$  by  $b(C)$  and  $\neg b(C)$ , respectively. Also, we denote the value of a variable  $x$  resulting after executing all events in a cut  $C$  by  $x(C)$ .

In this paper, we focus on a special but important class of predicates called *locally stable predicates* [17]. A predicate is *stable* if once the system reaches a global state where the predicate holds, the predicate holds in all future global states as well.

**Definition 1 (stable predicate).** *A predicate  $b$  is stable if it stays true once it becomes true. Formally,  $b$  is stable if for all consistent cuts  $C$  and  $D$ ,*

$$b(C) \wedge (C \subseteq D) \Rightarrow b(D)$$

An example of a stable predicate is termination (of a distributed computation) which is expressed as: “all processes are passive” and “all channels are empty”. Another important example of a stable predicate is deadlock which occurs when two or more processes are involved in some sort of “circular” wait. (Deadlock is stable under all request models.) A stable predicate is said to be *locally stable* if once the predicate becomes true, no variable involved in the

predicate changes its value thereafter. For a predicate  $b$ , let  $vars(b)$  denote the set of variables on which  $b$  depends.

**Definition 2 (locally stable predicate [17]).** *A stable predicate  $b$  is locally stable if no process involved in the predicate can change its state relative to  $b$  once  $b$  holds. Formally,  $b$  is locally stable if for all consistent cuts  $C$  and  $D$ ,*

$$b(C) \wedge (C \subseteq D) \Rightarrow \langle \forall x \in vars(b) :: x(C) = x(D) \rangle$$

Intuitively, once a locally stable predicate becomes true, not only does the value of the predicate stay the same—which is true, but the values of all variables involved in the predicate stay the same as well. In this paper, we distinguish between property and predicate. A predicate is a *concrete formulation* of a property in terms of program variables and processors states. In general, there is more than one way to formulate a property. For example, the mutual exclusion property, which states that there is at most one process in its critical section at any time, can be expressed in the following ways.

1.  $\bigwedge_{1 \leq i < j \leq n} (\neg cs_i \vee \neg cs_j)$ , where  $cs_i$  is true if and only if process  $p_i$  is in its critical section.
2.  $(\sum_{i=1}^n cs_i) \leq 1$ , where  $cs_i$  is 1 if and only if process  $p_i$  is in its critical section and is 0 otherwise.

Local stability, unlike stability, depends on the particular formulation of a property. It is possible that one formulation of a property is locally stable while the other is not. For instance, consider the property “the global virtual time of the system is at least  $k$ ”, which is abbreviated as  $GVT \geq k$  [19]. The property “ $GVT \geq k$ ” is true if and only if the local virtual time of each processes is at least  $k$  and there is no message in transit with timestamp less than  $k$ . Let  $lvt_i$  denote the local clock of process  $p_i$ . Also, let  $sent(i, j; k)$  denote the number of messages that process  $p_i$  has sent to process  $p_j$  so far whose timestamp is at most  $k$ . Likewise, let  $rcvd(i, j; k)$  denote the number of messages that process  $p_i$  has received from process  $p_j$  so far whose timestamp is at most  $k$ . The property  $GVT \geq k$  can be expressed as:

$$GVT > k \equiv \left( \bigwedge_{1 \leq i \leq n} lvt_i > k \right) \wedge \left( \bigwedge_{1 \leq i, j \leq n} sent(i, j; k) = rcvd(j, i; k) \right)$$

The above formulation of the property  $GVT \geq k$  is not locally stable because local clock of a process may change even after the predicate has become true. However, we can define an auxiliary variable  $a_i$  which is true if and only if  $lvt_i \geq k$ . An alternative formulation of the property  $GVT \geq k$  is:

$$GVT \geq k \equiv \left( \bigwedge_{1 \leq i \leq n} a_i \right) \wedge \left( \bigwedge_{1 \leq i, j \leq n} sent(i, j; k) = rcvd(j, i; k) \right)$$

Unlike the first formulation, the second formulation is actually locally stable. We say that a property is locally stable if there is at least one predicate representing the property that is locally stable. The complexity of determining whether a locally stable formulation for a given stable property exists is an open

problem. Termination, deadlock of a subset of processes (under single, AND, OR and  $k$ -out-of- $n$  request models) and global virtual time exceeding a given value can all be expressed as locally stable predicates.

### 3 The Algorithm

In this section, we describe an on-line algorithm to detect a locally stable predicate, that is, to determine whether a locally stable predicate has become true in a computation in progress. A general algorithm for detecting a stable predicate is to repeatedly compute consistent snapshots (or consistent cuts) of the computation and evaluate the predicate for these snapshots until the predicate becomes true. More efficient algorithms have been developed for detecting special cases of stable predicates such as termination and deadlock. Specifically, it has been shown that to detect many stable predicates, including termination and deadlock, it is not necessary for snapshots to be consistent. In this paper, we show that *any* locally stable predicate can be detected by repeatedly taking *possibly inconsistent* snapshots of the underlying computation.

#### 3.1 The Main Idea

The main idea is to take snapshots of the computation in such a manner that there is at least one consistent snapshot lying between any two consecutive snapshots. To that end, we generalize the notion of consistent cut to the notion of *consistent interval*.

**Definition 3 (interval).** *An interval  $[C, D]$  is a pair of possibly inconsistent cuts  $C$  and  $D$  such that  $C \subseteq D$ .*

An interval is said to be consistent if it contains at least one consistent cut.

**Definition 4 (consistent interval).** *An interval  $[C, D]$  is said to be consistent if there exists a consistent cut  $G$  such that  $C \subseteq G \subseteq D$ .*

Note that an interval  $[C, C]$  is consistent if and only if  $C$  is a consistent cut. Next, we give the necessary and sufficient condition for an interval to be consistent.

**Theorem 1.** *An interval  $[C, D]$  is consistent if and only if all events that happened-before some event in  $C$  belong to  $D$ . Formally,  $[C, D]$  is consistent if and only if the following holds:*

$$\langle \forall e, f :: (e \rightarrow f) \wedge (f \in C) \Rightarrow e \in D \rangle \quad (1)$$

Observe that when  $C = D$ , the necessary and sufficient condition for an interval to be consistent reduces to the definition of a consistent cut. Now, consider a consistent interval  $[C, D]$ . Suppose there is no change in the value of any variable in  $vars(b)$  between  $C$  and  $D$ . We say that the interval  $[C, D]$  is

*quiescent* with respect to  $b$ . Clearly, in this case, for every variable  $x \in \text{vars}(b)$ ,  $x(C) = x(D) = x(G)$ . This implies that  $b(G) = b(C) = b(D)$ . In other words, in order to compute the value of the predicate  $b$  for the consistent cut  $G$ , we can instead evaluate  $b$  for either endpoint of the interval, that is, cut  $C$  or cut  $D$ . In case  $b$  is a stable predicate and  $b(D)$  evaluates to true, we can safely conclude that  $b$  has indeed become true in the underlying computation. Formally,

**Theorem 2.** *If an interval  $[C, D]$  is consistent as well as quiescent with respect to a predicate  $b$ , then*

$$b(D) \Rightarrow \langle \exists G : G \text{ is a consistent cut} : b(G) \rangle$$

Based on the idea described above, an algorithm for detecting a locally stable predicate can be devised as follows. Repeatedly compute possibly inconsistent snapshots of the computation in such a way that every pair of *consecutive* snapshots forms a consistent interval. After each snapshot is recorded, test whether any of the relevant variables—on which the predicate depends—has undergone a change since the last snapshot was taken. In case the answer is “no”, evaluate the predicate for the current snapshot. If the predicate evaluates to true, then, using Theorem 2, it can be deduced that the computation has reached a state in which the predicate holds, and the detection algorithm terminates with “yes”. Otherwise, repeat the above steps for the next snapshot and so on.

Theorem 2 establishes that the algorithm is *safe*, that is, if the algorithm terminates with answer “yes”, then the predicate has indeed become true in the computation. We need to show that the algorithm is also *live*, that is, if the predicate has become true in the computation, then the algorithm terminates eventually with answer “yes”. To establish liveness, we use the fact that the predicate is locally stable, which was not required to prove safety. Suppose the predicate  $b$ , which is locally stable, has become true in the computation. Therefore there exists a consistent cut  $G$  of the computation that satisfies  $b$ . Let  $C$  and  $D$  with  $C \subseteq D$  be two snapshots of the computation taken after  $G$ . In other words,  $G \subseteq C \subseteq D$ . Since  $b$  is a locally stable predicate and  $b(G)$  holds, no variable in  $\text{vars}(b)$  undergoes a change in its value after  $G$ . This implies that the values of all the variables in  $\text{vars}(b)$  for  $D$  is same as that for  $G$  and therefore  $D$  satisfies  $b$  as well. Formally,

**Theorem 3.** *Given an interval  $[C, D]$ , a locally stable predicate  $b$  and a consistent cut  $G$  such that  $G \subseteq C$ ,*

$$b(G) \Rightarrow ([G, D] \text{ is quiescent with respect to } b) \wedge b(D)$$

Observe that if  $[G, D]$  is quiescent with respect to  $b$  then so is  $[C, D]$ . The algorithm, on detecting that no relevant variable has undergone a change in the interval  $[C, D]$ , evaluates  $b$  for  $D$ . In this case,  $b(D)$  evaluates to true and, as a result, the algorithm terminates with answer “yes”.

### 3.2 Implementation

To implement the detection algorithm described in the previous section, two issues need to be addressed. First, how to ensure that every pair of consecutive

snapshots forms a consistent interval. Second, how to detect that no relevant variable has undergone a change in a given interval, that is, all relevant variables have reached a state of quiescence. We next discuss solutions to both the problems.

**Ensuring Interval Consistency using Barrier Synchronization:** First, we give a condition that is stronger than the condition (1) given in Theorem 1 in the sense that it is sufficient but not necessary for a pair of cuts to form a consistent interval. The advantage of this condition is that it can be easily implemented using only *control messages* without altering messages generated by the underlying computation, hereafter referred to as *application messages*. To that end, we define the notion of *barrier synchronized interval*. Intuitively, an interval  $[C, D]$  is barrier synchronized if it is not possible to move beyond  $D$  on any process until all events in  $C$  have been executed.

**Definition 5 (barrier synchronized interval).** *An interval  $[C, D]$  is barrier synchronized if every event contained in  $C$  happened-before every event that does not belong to  $D$ . Formally,*

$$\langle \forall e, f :: (e \in C) \wedge (f \notin D) \Rightarrow e \rightarrow f \rangle \quad (2)$$

Next, we show that a barrier synchronized interval is also consistent.

**Lemma 4 (barrier synchronization  $\Rightarrow$  consistency).** *If an interval is barrier synchronized then it is also consistent.*

It can be verified that when  $C = D$ , the notion of barrier synchronized interval reduces to the notion of barrier synchronized cut, also known as *inevitable global state* [20]. Now, to implement the algorithm described in the previous section, we use a *monitor* which periodically records snapshots of the underlying computation. One of the processes in the system can be chosen to act as a monitor. In order to ensure that every pair of consecutive snapshots is barrier synchronized, the monitor simply needs to ensure that the protocol for recording the next snapshot is initiated only *after* the protocol for recording the current snapshot has terminated. Recording a snapshot basically requires the monitor to collect local states of all processes. Many approaches can be used depending upon the communication topology and other factors. For instance, the monitor can broadcast a message to all processes requesting them to send their local states. A process, on receiving message from the monitor, sends its (current) local state to the monitor [6]. Alternatively, processes in the system can be arranged to form a logical ring. The monitor uses a token (sometimes call a probe) which circulates through the entire ring gathering local states on its way [9, 10, 17]. Another approach is to impose a spanning tree on the network with the monitor acting as the root. In the first phase, starting from the root node, control messages move downward all the way to the leaf nodes. In the second phase, starting from leaf nodes, control messages move upward to the root node collecting local states on their way [19]. (The local states are recorded in the second

phase and not in the first phase.) Hereafter, we refer to the three approaches discussed above as *broadcast-based*, *ring-based* and *tree-based*, respectively. In all the three approaches, recording of a local state can be done in a lazy manner [10]. In lazy recording, a process postpones recording its local state until its current local state is such that it does not preclude the (global) predicate from becoming true. For instance, in termination detection, a process which is currently *active* can postpone recording its local state until it becomes *passive*.

Let a *session* correspond to taking a single snapshot of the computation. For the  $k^{\text{th}}$  session, let  $S_k$  refer to the snapshot computed in the session, and let  $start_k$  and  $end_k$  denote the events on the monitor that correspond to the beginning and end of the session. All the above approaches ensure the following:

$$\langle \forall e : e \in \text{frontier}(S_k) : e \rightarrow end_k \rangle \wedge \langle \forall f : f \in \text{frontier}(S_{k+1}) : start_{k+1} \rightarrow f \rangle$$

Since sessions do not overlap,  $end_k \rightarrow start_{k+1}$ . This implies that:

$$\langle \forall e, f :: (e \in \text{frontier}(S_k)) \wedge (f \in \text{frontier}(S_{k+1})) \Rightarrow e \rightarrow f \rangle \quad (3)$$

It can be easily verified that (3) implies (2). Note that non-overlapping of sessions is only a sufficient condition for interval consistency and not necessary. It is possible to ensure interval consistency even when sessions overlap. However, application messages need to be modified to carry control information.

**Detecting Interval Quiescence using Dirty Bits:** To detect whether one or more variables have undergone a change in their values in a given interval, we use *dirty bits*. Specifically, we associate a dirty bit with each variable whose value the predicate depends on. Sometimes, it may be possible to associate a single dirty bit with a set of variables or even the entire local state. Initially, each dirty bit is in its *clean state*. Whenever there is a change in the value of a variable, the corresponding dirty bit is set to an *unclean state*. When a local snapshot is taken (that is, a local state is recorded), all dirty bits are also recorded along with the values of all the variables. After the recording, all dirty bits are reset to their clean states. Clearly, an interval  $[C, D]$  is quiescent if and only if all dirty bits in  $D$  are in their clean states.

In case multiple monitors are used to achieve fault-tolerance, a separate set of dirty bits has to be maintained for each monitor. This is to prevent snapshots protocols initiated by different monitors from interfering with each other; otherwise dirty bits may be reset incorrectly.

**Combining the Two:** To detect a locally stable predicate, the monitor executes the following steps.

1. Compute a snapshot of the computation.
2. Test whether all dirty bits in the snapshot are in their clean states. If not, go to the first step.
3. Evaluate the predicate for the snapshot. If the snapshot does not satisfy the predicate, then go to the first step.

The basic algorithm can be further optimized. In the ring-based approach, the process currently holding the token can discard the token if the local states gathered so far indicate that the global predicate has not become true. For example, this can happen during termination detection when the token reaches a process with one or more dirty bits in their unclean states. The process discarding the token can either inform the monitor or become the new monitor itself and initiate the next session for recording a snapshot. When a session is *aborted early* in this manner, only a subset of processes would have recorded their local states and have their dirty bits reset. In this case, the global snapshot for a session, even if it is aborted early, can be taken to be the collection of *last recorded* local states on all processes.

## 4 Performance Analysis

We now analyze the performance of the three concrete variants of our detection algorithm, namely broadcast-based, ring-based and tree-based. We evaluate the three approaches with respect to the following criteria:

- **Message Complexity:** It refers to the number of (control) messages generated by the algorithm. These messages are in addition to the application messages generated by the underlying computation.
- **Message Overhead:** It refers to the maximum size of a control message expressed in number of bits.
- **Detection Latency (or Delay):** It refers to the time, measured as the number of message hops, elapsed between when the predicate becomes true to when the detection algorithm terminates.
- **Process Load:** It refers to the number of control messages exchanged—sent or received—by a process.

Let the space-complexity of recording a local state be  $O(s)$  bits.

**Broadcast-based approach:** For this approach, the message complexity per session is  $2(n - 1)$ , where  $n$  is the number of processes, and the message overhead for a control message is  $O(s)$ . Once the predicate becomes true, the algorithm requires at most two more sessions to terminate after the current session has terminated. This is because, after the current session is over, the next session will reset all dirty bits and the session after that will detect the predicate. This translates into  $O(1)$  message hops. The monitor is involved in  $2(n - 1)$  message exchanges per session; it sends  $n - 1$  messages and receives  $n - 1$  messages. All other processes are involved in two message exchanges per session; each one of them receives one message and sends one message. Therefore the broadcast-based approach is highly centralized in nature and as such is not suitable for large systems because the monitor may get swamped by messages from other processes.

**Ring-based approach:** For this approach, the message complexity per session is  $n$  and the message overhead for a control message is  $O(ns)$ . Depending on the

property being detected, however, the message overhead may be much lower. For example, for termination detection, it is not necessary to store the local state of each process that has been visited by the token separately. It is sufficient to have one bit to indicate whether all dirty bits seen so far are in their clean states, one bit to indicate whether all processes seen so far are passive, and one integer to store the message deficit—the number of messages sent minus the number of messages received summed over all processes visited so far [9]. The detection latency is two sessions after the current session terminates, which translates into  $O(n)$  message hops. This approach is attractive due to its distributed nature because each process is involved in two message exchanges per session; it receives one message and sends one message.

**Tree-based approach:** This approach lies in between broadcast-based and ring-based approaches. The message complexity per session is  $2(n-1)$  and the message overhead for a control message is  $O(ns)$ . Again, depending on the predicate, the message overhead may be much lower. As in other two approaches, the detection latency is two sessions after the current session terminates. Therefore the detection latency in terms of message hops is  $O(h)$ , where  $h$  is the height of the tree. For a process  $p$ , let  $\text{degree}(p)$  denote the number of neighbors of  $p$  in the spanning tree. For example, if  $p$  is a leaf node then  $\text{degree}(p) = 1$ . Clearly, process  $p$  exchanges  $2 * \text{degree}(p)$  messages per session; it sends  $\text{degree}(p)$  messages and receives  $\text{degree}(p)$  messages.

## 5 Discussion

Marzullo and Sabel give an algorithm for detecting a locally stable predicate using the notion of *weak vector clock* [17]. A weak vector clock, unlike the Fidge/Mattern’s vector clock [21, 22], is updated only when an event that is “relevant” with respect to the predicate is executed. Whenever a process sends a message, it piggybacks the current value of its local (weak) vector clock on the message. Thus Marzullo and Sabel’s algorithm requires application messages to be modified to carry a vector timestamp of size  $n$ , where  $n$  is the number of processes.

Ho and Ramamoorthy give a two-phase protocol to detect a deadlock in a distributed database system using AND request model [6]. Their two-phase approach is similar to our (broadcast-based) approach in the sense that the snapshots computed in the two phases form a consistent interval. However, their methodology for detecting what part of a process state is quiescent (and what is not) in an interval is flawed. Specifically, their quiescence detection approach works only if status tables maintain information about transactions (and not processes) and transactions follow two-phase locking discipline.

In [23], Kshemkalyani and Singhal propose modification to the Ho and Ramamoorthy’s two phase deadlock detection algorithm and prove that their modified algorithm is correct. They also discuss how their modified two-phase algorithm can be used for detecting a general stable property. Our work is different from Kshemkalyani and Singhal’s work in the following ways. First, their work

is based on the notion of real-time, whereas ours is based on the notion of Lamport's happened-before relation. Second, Kshemkalyani and Singhal claim that their algorithm can be used for general stable property detection. However, as we show in this paper, two phase algorithm can only be used to detect a locally stable predicate. For a stable predicate that is not locally stable, the algorithm is guaranteed to be safe but not live.

Termination detection algorithms by Safra [9] and Mattern *et al* [10] are similar to our ring-based approach. In Safra's algorithm, a *color* is associated with every machine and the token. A machine turns black when it transitions from active to passive (Rule 3'). When a token visits a black machine, it also turns black (Rule 4). A black machine holding the token turns white after sending the token to its neighbouring machine (Rule 7). Termination is detected when, after one full circulation, all machines were in their passive states, the message deficit is zero, and the token stays white. Mattern *et al* associate a *sticky flag* with every process. A sticky flag normally tracks the state of a process. However, when a process transitions from active to passive, the flag sticks to active until a control message "unsticks" it. Termination is detected when, after one full circulation, all processes along with their sticky flags were in their passive states, and the message deficit is zero. Color and sticky flag play the same role in the two algorithms as dirty bits in ours; both are used to detect if a process went through some activity since last recording its local state. Besides the above two examples, there are several other termination detection algorithms that can be viewed as special cases of our approach for detecting a locally stable predicate such as Mattern's four counter algorithm [8] and H elary and Raynal's algorithm based on "continuously-passive" flag [11].

Our work is different in the sense that our algorithm is more general and can be used to detect *any* locally stable predicate, and not just termination and deadlock.

## 6 Conclusion and Future Work

In this paper, we give an efficient algorithm to detect a locally stable predicate based on repeatedly taking (possibly inconsistent) snapshots of the computation in a certain manner. Our algorithm uses only control messages and thus application messages do not need to be modified to carry any control information. It also unifies several known algorithms for detecting two important locally stable predicates, namely termination and deadlock.

At present, we assume that the system is not subject to any failures. In a real world, however, failures do occur and one or more processes may crash. We plan to modify our detection algorithm to work in a faulty environment when one or more processes can fail by crashing. Our algorithm also assumes that it is possible to monitor changes in the values of the relevant variables efficiently. This can be accomplished in two ways. In the first approach, application program is modified such that whenever a relevant variable is assigned a new value, the detection algorithm is informed of the change. In the second approach, which is

more desirable, monitoring is done in a transparent manner without modifying the underlying program. While most debuggers such as gdb already have such a capability, their approach is very inefficient. As future work, we plan to investigate efficient ways for monitoring an application program in a transparent manner.

## References

1. Chandy, K.M., Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems* **3** (1985) 63–75
2. Lai, T.H., Yang, T.H.: On Distributed Snapshots. *Information Processing Letters (IPL)* **25** (1987) 153–158
3. H elary, J.M., Jard, C., Plouzeau, N., Raynal, M.: Detection of Stable Properties in Distributed Applications. In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. (1987) 125–136
4. Acharya, A., Badrinath, B.R.: Recording Distributed Snapshots Based on Causal Order of Message Delivery. *Information Processing Letters (IPL)* **44** (1992) 317–321
5. Alagar, S., Venkatesan, S.: An Optimal Algorithm for Recording Snapshots using Casual Message Delivery. *Information Processing Letters (IPL)* **50** (1994) 311–316
6. Ho, G.S., Ramamoorthy, C.V.: Protocols for Deadlock Detection in Distributed Database Systems. *IEEE Transactions on Software Engineering* **8** (1982) 554–557
7. Misra, J.: Detecting Termination of Distributed Computations using Markers. In: *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. (1983) 290–294
8. Mattern, F.: Algorithms for Distributed Termination Detection. *Distributed Computing (DC)* **2** (1987) 161–175
9. Dijkstra, E.W.: Shmuel Safra’s Version of Termination Detection. EWD Manuscript 998. Available at <http://www.cs.utexas.edu/users/EWD> (1987)
10. Mattern, F., Mehl, H., Schoone, A., Tel, G.: Global Virtual Time Approximation with Distributed Termination Detection Algorithms. Technical Report RUU-CS-91-32, University of Utrecht, The Netherlands (1991)
11. H elary, J.M., Raynal, M.: Towards the Construction of Distributed Detection Programs, with an Application to Distributed Termination. *Distributed Computing (DC)* **7** (1994) 137–147
12. Brzezinski, J., H elary, J.M., Raynal, M., Singhal, M.: Deadlock Models and a General Algorithm for Distributed Deadlock Detection. *Journal of Parallel and Distributed Computing (JPDC)* **31** (1995) 112–125
13. Demirbas, M., Arora, A.: An Optimal Termination Detection Algorithm for Rings. Technical Report OSU-CISRC-2/00-TR05, The Ohio State University (2000)
14. Stupp, G.: Stateless Termination Detection. In: *Proceedings of the 16th Symposium on Distributed Computing (DISC)*, Toulouse, France (2002) 163–172
15. Khokhar, A.A., Hambruch, S.E., Kocalar, E.: Termination Detection in Data-Driven Parallel Computations/Applications. *Journal of Parallel and Distributed Computing (JPDC)* **63** (2003) 312–326
16. Mahapatra, N.R., Dutt, S.: An Efficient Delay-Optimal Distributed Termination Detection Algorithm. To Appear in *Journal of Parallel and Distributed Computing (JPDC)* (2004)

17. Marzullo, K., Sabel, L.: Efficient Detection of a Class of Stable Properties. *Distributed Computing (DC)* **8** (1994) 81–91
18. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)* **21** (1978) 558–565
19. Tel, G.: *Introduction to Distributed Algorithms*. Second edn. Cambridge University Press (US Server) (2000)
20. Fromentin, E., Raynal, M.: Inevitable Global States: A Concept to Detect Unstable Properties of Distributed Computations in an Observer Independent Way. In: *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing (SPDP)*. (1994) 242–248
21. Mattern, F.: Virtual Time and Global States of Distributed Systems. In: *Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG)*, Elsevier Science Publishers B. V. (North-Holland) (1989) 215–226
22. Fidge, C.: Logical Time in Distributed Computing Systems. *IEEE Computer* **24** (1991) 28–33
23. Kshemkalyani, A., Singhal, M.: Correct Two-phase and One-phase Deadlock Detection Algorithms for Distributed Systems. In: *Proceedings of the IEEE Symposium on Parallel and Distributed Processing (SPDP)*. (1990) 126–129