Efficient detection of a locally stable predicate in a distributed system ☆

Ranganath Atreya[a,1], Neeraj Mittal[b,∗], Ajay D. Kshemkalyani[c], Vijay K. Garg[d,2], Mukesh Singhal[e]

[a]*Web Services Technologies, Amazon.com, Inc., Seattle, WA 98101, USA*
[b]*Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA*
[c]*Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607, USA*
[d]*Electrical and Computer Engineering Department, The University of Texas at Austin, Austin, TX 78712, USA*
[e]*Department of Computer Science, The University of Kentucky, Lexington, KY 40506, USA*

Abstract

We present an efficient approach to detect a locally stable predicate in a distributed computation. Examples of properties that can be formulated as locally stable predicates include termination and deadlock of a subset of processes. 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. The worst-case message complexity of our algorithm is $O(n(m + 1))$, where n is the number of processes in the system and m is the number of events executed by the underlying computation. We show that, in practice, its message complexity should be much lower than its worst-case message complexity. The detection latency of our algorithm is $O(d)$ time units, where d is the diameter of communication topology. Our approach also *unifies* several known algorithms for detecting termination and deadlock. We also show that our algorithm for detecting a locally stable predicate can be used to efficiently detect a stable predicate that is a monotonic function of other locally stable predicates.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Monitoring distributed computation; Stable property detection; Termination detection; Deadlock detection; Global virtual time computation; Inconsistent snapshots

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 said to be stable if it stays true 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 [10,31,18,1,2]. These algorithms can be broadly classified into four categories. They either require sending a control message along every channel in the system [10] or rely on piggybacking control information on application messages [31] or assume that messages are delivered in causal order [1,2] or are inhibitory in nature [18]. 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., [20,41,37,14,38,22,40,19,8,13,47,24,34,44,43]).

Termination and deadlock are examples of stable properties that can be formulated as *locally stable predicates* [36]. 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 taking possibly inconsistent snapshots of the computation in a certain manner. Our algorithm does not inhibit any event of the underlying computation nor does it require messages to be delivered in a certain order. Unlike Marzullo and Sabel's algorithm for detecting a locally stable predicate [36], no control information is required to be piggybacked on application messages and, therefore, application messages do not need to be modified at all. This saves on the message size and, therefore, network bandwidth, and also avoids the overheads of another software layer to parse each and every application message. Another advantage of our approach is that it does not require snapshots to be consistent, and hence it is not necessary for processes to *coordinate* their actions when taking a snapshot.

The worst-case message complexity of our algorithm is $O(n(m+1))$, where n is the number of processes in the system and m is the number of events executed by the computation before the predicate becomes true. We show that, in practice, its average-case message complexity should be much lower than its worst-case message complexity. The detection latency of our algorithm is $O(d)$ time units, where d is the diameter of the communication topology.

Our general algorithm for detecting a locally stable predicate also *unifies* several known algorithms for detecting termination and deadlock [20,37,14,40,19]. Some of the examples include Safra's color-based algorithm [14], Mattern's four-counter algorithm [37] and Mattern et al.'s sticky-flag algorithm [40] for termination detection, and Ho and Ramamoorthy's two-phase algorithm [20] for deadlock detection. All of these algorithms can be derived as special cases of the approach given in this paper. Therefore, this paper presents a *unifying* framework for understanding and describing various termination and deadlock detection algorithms. We argue that the performance of the presented algorithm is asymptotically no worse than that of the specialized algorithms, and in many cases, it performs better. We also instantiate our general algorithm to derive an efficient deadlock detection algorithm whose performance is comparable with that of existing deadlock detection algorithms. Further, we show that our algorithm can be used to efficiently detect a stable predicate that can be expressed as a *monotonic function* of other locally stable predicates. Note that the two-phase deadlock detection algorithm as described in [20] is actually flawed [23] but can be corrected using the ideas given in this paper. A correct version of the two-phase deadlock detection algorithm can be found in [26]. Finally, we discuss how our approach can be extended to work in the presence of process crashes.

A special feature of our algorithm is that it does not require application messages to be modified to assist the detection algorithm, unlike many other algorithms (e.g., [22,37,19,8,47,34]).

The algorithm detects locally stable predicates solely by monitoring changes in the values of the relevant variables. Although this may require modification of the application program, it cannot be considered as an extra overhead because most algorithms for predicate detection also monitor changes in the values of relevant variables and, therefore, require the application program to be modified to aid in the detection process (e.g., [20,22,37,40,19,8,47,34]).

The paper is organized as follows. Section 2 describes the system model and notations used in this paper. A basic algorithm for detecting a locally stable predicate is proposed in Section 3 after presenting the main ideas behind the algorithm. Section 4 gives a complexity analysis of the algorithm. Section 5 gives an improved algorithm that has a bounded worst-case complexity. Section 6 describes three applications of our algorithm. Specifically, we describe how a stable predicate, expressed as a monotonic function of other locally stable predicates, can be detected efficiently using our approach. We also show that many algorithms for detecting termination and deadlock can be viewed as special cases of our algorithm. We discuss modifications to our algorithm to make it fault-tolerant in Section 7. We discuss the related work in Section 8. Finally, Section 9 concludes the paper and outlines directions for future research.

2. Model and notation

In this section we formally describe the model and notations used in this paper.

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 are bidirectional and 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. Let $proc(e)$ denote the process on which event e is executed. 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 →, is given by the Lamport's *happened-before relation* (also known as *causality relation*) [32] 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 events, respectively, of the same message then e happened-before f.

Intuitively, the Lamport's happened-before relation captures the maximum amount of information that can be deduced about the 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* or *global snapshot*, 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 the 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,

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

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

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

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,

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

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 that cut. In other words, only those cuts which are consistent can possibly occur during an execution. In this paper, we use the terms "snapshot" and "cut" interchangeably.

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, $b(C) =$ true. Hereafter, we abbreviate expressions $b(C) =$ true and $b(C) =$ 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* [36]. 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,

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

Termination of a distributed computation, expressed as "all processes are passive" and "all channels are empty", is an example of a stable predicate. A deadlock in a distributed database system—which occurs when two or more processes are involved in some sort of circular wait—and inaccessibility of an object can also be expressed as stable predicates. 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, Marzullo and Sabel [36]*). 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. $\bigwedge_{1 \leqslant i < j \leqslant n} (\neg cs_i \vee \neg cs_j)$, where cs_i is true if and only if process p_i is in its critical section.
2. $(\sum_{i=1}^{n} cs_i) \leqslant 1$, 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 has advanced beyond k", which is abbreviated as $GVT > k$ [49]. The property "$GVT > k$" is true if and only if the local virtual time of each process has advanced beyond k and there is no message in transit with timestamp at most k. Let lvt_i denote the local clock of process p_i. Also, let $sent(i, j; k)$ denote the number of messages with timestamp at most k that process p_i has sent to process p_j so far. Likewise, let $rcvd(i, j; k)$ denote the number of messages with timestamp at most k that process p_i has received from process p_j so far. The property $GVT > k$ can be expressed as

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

The above formulation of the property $GVT > k$ 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 $lvt_i > k$. An alternative formulation of the property $GVT > k$ is

$$GVT > k \equiv \left(\bigwedge_{1 \leqslant i \leqslant n} a_i \right)$$
$$\bigwedge \left(\bigwedge_{1 \leqslant i, j \leqslant n} sent(i, j; k) = rcvd(j, i; k) \right).$$

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 way to formulate the property such that the formulation corresponds to a locally stable predicate. 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.

Remark 1. A deadlock exists in the system if and only if there exists a subset of processes such that the processes in the subset are involved in some sort of circular wait. The exact nature of the "circular wait" depends on the request model used. For example, in the AND request model, a circular wait corresponds to a cycle in the wait-for graph [21,25,45], whereas, in the OR request model, it corresponds to a knot in the wait-for graph [21,25,45]. (In the AND request model, a process may specify one or more resources at the time of the request, and can proceed only if it receives permission to access all the requested resources. In the OR request model, on the other hand, a process may specify one or more resources at the time of the request, and can proceed if it receives permission to access any one of the requested resources.)

Consider the AND request model. For a subset of processes Q, where $Q \subseteq P$, let $WFG(Q)$ denote the wait-for graph that exists among processes in Q. Then,

$$circular\text{-}wait(Q) \equiv WFG(Q) \text{ forms a simple cycle.}$$

Clearly, the property "$WFG(Q)$ forms a simple cycle" can be formulated as a locally stable predicate. Specifically, for a process $p_i \in Q$, let wf_i^Q denote the set of processes in Q that process p_i is waiting-for. Once the property "$WFG(Q)$ forms a simple cycle" becomes true, none of the variables wf_i^Q for each $p_i \in Q$ change their values after that. The deadlock property under AND request model can now be expressed as

$$deadlock(P) \equiv \langle \exists Q : Q \subseteq P : circular\text{-}wait(Q) \rangle.$$

Note that $deadlock(P)$ by itself cannot be expressed as a locally stable predicate. This is because, even if some processes in P are involved in a deadlock, other processes in P may continue to execute and therefore their variables may continue to change their values. However, it can be expressed as a disjunction of locally stable predicates, where each disjunct is of the form $circular\text{-}wait(Q)$ for some Q. We refer to such a predicate as monotonically decomposable stable predicate and we discuss it in greater detail later in Section 6. A similar observation can be made for deadlock under other request models.

3. An algorithm for detecting a locally stable predicate

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 certain stable properties such as termination and deadlock. Specifically, it has been shown that to detect many stable properties, including termination and deadlock, it is not necessary for snapshots to be consistent. In this section, 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 $C \subseteq D$.

We next formally define what it means for an interval to be consistent.

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*:

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

Proof. First, assume that $[C, D]$ is a consistent interval. This implies that there exists a consistent cut G such that $C \subseteq G \subseteq D$. Pick any events e and f such that $f \in C$ and $e \to f$. Since $f \in C$ and $C \subseteq G$, we get that $f \in G$. From the fact that G is consistent, we get that $e \in G$. But $e \in G$ and $G \subseteq D$ imply that $e \in D$.

Conversely, assume that (1) is true. We define the cut G as consisting of all events in C and those that happened-before them. Formally,

$$G \triangleq \{e \mid \exists f \in C : e = f \text{ or } e \to f\}.$$

Evidently, from the definition of G, $C \subseteq G$. To show that $G \subseteq D$, consider an event $e \in G$. By definition of G, there exists an event $f \in C$ such that either $e = f$ or $e \to f$. In the first case, $e \in D$ because $C \subseteq D$. In the second case, $e \in D$ because (1) holds. Now, we only need to show that G is consistent. Pick any events u and v such that $u \to v$ and $v \in G$. From the definition of G, there exists an event $f \in C$ such that either $v = f$ or $v \to f$. In either case, $u \to f$. Since $f \in C$ and $u \to f$, by definition of G, we get that $u \in G$. Hence G is consistent. $\square$

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. In that case, we say that the interval $[C, D]$ is *quiescent* with respect to b. Consider a consistent cut G that lies between C and D. Clearly, for every variable $x \in vars(b)$, $x(C) = x(D) = x(G)$. 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 $vars(b)$ undergoes a change in its value after G. This implies that the values of all the variables in $vars(b)$ for D is same as that for G and therefore D satisfies b as well. Formally,

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

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

Intuitively, Theorem 3 implies that, for an algorithm to be live, it is sufficient for the algorithm to take at least two snapshots forming a consistent interval after the predicate has become true.

Theorem 4. *Consider a locally stable predicate* b. *If an algorithm takes at least two cuts* C *and* D *after* b *has become true such that interval* $[C, D]$ *is consistent*, *then the algorithm eventually detects that* b *has become true*.

Proof. Since C and D are taken after b has become true, there exists a consistent cut G that satisfies b such that $G \subseteq C$. From Theorem 3, interval $[G, D]$ is quiescent with respect to b, and, therefore, so is the interval $[C, D]$. From Theorem 2, b evaluates to true for D. $\square$

We next describe how to ensure that a pair of consecutive snapshots form a consistent interval and how to detect that the interval they form is quiescent.

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 problems.

3.2.1. 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 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,

$$\langle \forall e, f :: (e \in C) \wedge (f \notin D) \Rightarrow e \to f \rangle. \tag{2}$$

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

Lemma 5 (*barrier synchronization ⇒ consistency*). *If an interval is barrier synchronized, then it is also consistent.*

Proof. The proof is by contradiction. Assume that an interval $[C, D]$ is barrier synchronized but is not consistent. Therefore there exist events u and v with $u \to v$ such that $v \in C$ but $u \notin D$. Since $[C, D]$ is barrier synchronized, $v \to u$. However, $u \to v$ and $v \to u$ imply that $\to$ contains a cycle—a contradiction. □

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* [17]. 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 instance (of the snapshot algorithm) for recording the next snapshot is initiated only *after* the instance 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 [49]. For instance, the monitor broadcasts 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 [20]. Alternatively, processes in the sys-

tem can be arranged to form a logical ring. The monitor uses a token (sometimes called a probe) which circulates through the entire ring gathering local states on its way [14,40,36]. Another approach is to impose a spanning tree on the network with the monitor acting as the root. The monitor collects a snapshot using a combination of broadcast and convergecast on the spanning tree. Specifically, in the broadcast phase, starting from the root node, control messages move downward all the way to the leaf nodes. In the convergecast phase, starting from leaf nodes, control messages move upward to the root node collecting local states on their way [49]. (A process can record its local snapshot either during the broadcast phase or during the convergecast 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 [40]. 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*. Hereafter, unless otherwise specified, we assume that the tree-based approach is used for taking a snapshot. This is because, with the tree-based approach, no assumption has to be made about the communication topology (e.g., whether it is fully connected).

Let a *session* corresponds to taking a single snapshot of the computation. For the kth 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 the end of the session, respectively. All the above approaches ensure the following:

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

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

$$\langle \forall e, f :: (e \in frontier(S_k)) \wedge (f \in frontier(S_{k+1}))$$
$$\Rightarrow e \to f \rangle. \tag{3}$$

It can be easily verified that (3) implies (2). Therefore, we have,

Lemma 6. *If a new instance of snapshot algorithm is started only after the current instance of snapshot algorithm has finished, then the two snapshots form a consistent interval.*

Note that non-overlapping of sessions is a sufficient condition for interval consistency, but it is not necessary. It is possible to ensure interval consistency even when sessions overlap. However, application messages need to be modified to carry control information.

3.2.2. 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,

Lemma 7. *An interval* $[C, D]$ *is quiescent if and only if all dirty bits in D are in their clean states.*

For properties that may be evaluated on a partial state (e.g., a subset of processes are involved in a circular wait), it is more efficient for a process to record only the *quiescent part of its state* when taking a local snapshot. Specifically, instead of recording the values of all the variables along with the associated dirty bits, it may be more efficient to record the values of only those variables that have not undergone any change since the last local snapshot was recorded. The monitor, on collecting the quiescent part of the local state from each process, can evaluate the locally stable predicate on the collected state, if it is possible to do so.

3.2.3. Combining the two: the BasicLSPD algorithm

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.

We use BasicLSPD to refer to the algorithm described in this section.

3.2.4. Optimizing the algorithm

The basic algorithm BasicLSPD 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. This can happen, for example, 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. Complexity analysis

We analyze the performance of BasicLSPD with respect to three metrics: message complexity, detection latency and space complexity.

Message complexity measures the number of message exchanged by the predicate detection algorithm. Detection latency measures the time elapsed between when the predicate becomes true and when the algorithm detects that the predicate has become true. To measure detection latency, it is typically assumed that message transmission time is at most one time unit and message processing time is negligible [49,4]. (A similar assumption is made when computing time-complexity of an algorithm for an asynchronous distributed system as well.) Finally, space complexity measures the amount of space used by the predicate detection algorithm.

Message complexity: The worst-case message complexity of BasicLSPD is unbounded. This is because an adversary can force the monitor to initiate an *unbounded* number of instances of the snapshot algorithm before the monitor detects that the predicate has become true. However, we show that its message complexity should be much lower in practice.

Let n denote the number of processes in the system and d denote the diameter of the communication topology. We first calculate the average time an instance of a snapshot algorithm takes to execute, denoted by δ_s. Let δ_c denote the average channel delay. Unless otherwise stated, assume that message processing time is negligible and a snapshot is collected using the tree-based approach via a breadth-first-search (BFS) spanning tree. When using a tree-based approach to collect a snapshot of the system, messages are required to travel from the root node, which acts as a monitor, to leaf nodes and back. Therefore,

$$\delta_s \approx 2 d \delta_c.$$

Let δ_w denote the average time for which a monitor waits before initiating the next instance of a snapshot algorithm. We now compute the average number of events that are executed in the system between two consecutive initiations of a snapshot algorithm, denoted by η_e. Let δ_e denote the average delay between two consecutive events of a process. We have,

$$\eta_e \approx n \frac{\delta_s + \delta_w}{\delta_e}.$$

Let m denote the total number of events executed by the underlying computation before the predicate becomes true. We next compute the average number of instances of a snapshot algorithm initiated by the monitor, denoted by η_s. We have,

$$\eta_s \approx \frac{m}{\eta_e} + 2 \approx \frac{m}{n} \frac{\delta_e}{\delta_s + \delta_w} + 2.$$

The additional two snapshots are used to account for the fact that the monitor may have to take two snapshots of the system even after the predicate has become true before detecting that the predicate has indeed become true. With tree-based approach, each instance of a snapshot algorithm generates $2(n-1)$ messages. Therefore, the average number of messages

generated by BasicLSPD , denoted by η_m, is given by

$$\eta_m \approx 2\,(n-1)\,\eta_s \approx 2\,(n-1)\left(\frac{m}{n}\,\frac{\delta_e}{\delta_s+\delta_w}+2\right)$$
$$\approx \frac{2\,m\,\delta_e}{2\,d\,\delta_c+\delta_w}+4\,(n-1).$$

With $\delta_c = 5ms$, $\delta_e = 50ms$, $\delta_w = 50ms$ and $d = 25$, $\eta_m \approx \frac{m}{3}+4(n-1)$. We expect m to be much greater than n in practice because, in a realistic computation, each process should execute many events. Therefore, the average message complexity of BasicLSPD is strongly influenced by the multiplier factor for m, which is given by $\frac{2\,\delta_e}{2\,d\,\delta_c+\delta_w}$. It is important to note that the multiplier factor for m decreases as d increases.

An interesting question is: *how does our general algorithm for detecting any locally stable predicate compare with a more specialized algorithm for detecting a specific locally stable predicate, say, termination?* To our knowledge, if the communication topology is arbitrary, then all message-optimal termination detection algorithms are acknowledgment-based [15,9,43]; they generate an acknowledgment message for every application message generated by the underlying computation. As a result, their message complexity even in average case is at least $m+n-1$. (The additional $n-1$ messages are required in case the computation is non-diffusing, that is, any subset of processes may be active initially.) A similar argument can be made for deadlock detection algorithms in this class (e.g., [20,26]).

As our discussion illustrates, depending on the values of various parameters, the message-complexity of BasicLSPD, in practice, may be significantly lower than that of even more specialized detection algorithms even though its worst-case message complexity is unbounded. In computing the average message complexity, we assume that processes record their snapshots as soon as they learn that a new instance of the snapshot algorithm is in progress. If, on the other hand, processes record their snapshots in a lazy manner, then the average message complexity should be even lower. We also note here that we can restrict our algorithm to require participation by only those processes that are part of the global predicate that is being evaluated.

Detection latency: Once the predicate becomes true, from Theorem 2, the monitor needs to collect at most two snapshots of the system to detect that the predicate has become true. Therefore, the detection latency of BasicLSPD depends on the time-complexity of the snapshot algorithm. If a BFS spanning tree is used to collect a snapshot of the system, then the time-complexity of a snapshot algorithm is $O(d)$. Thus, the detection latency of BasicLSPD is also $O(d)$.

Space complexity: Let s denote the amount of space required to store a local snapshot of a single process. The monitor, in the worst-case, may have to store local snapshots of all processes explicitly before it can evaluate the predicate. Therefore, the space complexity of BasicLSPD is $O(n\,s)$ in the worst-case. In many cases, however, it may be possible to combine local snapshots of processes into a more succinct form, which can still be used to evaluate the predicate. For example, when detecting termination, it is sufficient for the monitor to know if one of the processes is active or one of the channels is empty.

It is not necessary to know the states of all processes and channels explicitly. Therefore, in the case of termination detection, space complexity of BasicLSPD may be as low as $O(1)$.

5. A worst-case message-complexity bounded algorithm

The algorithm BasicLSPD has one major drawback. Its worst-case message complexity is unbounded. We describe how to bound the worst-case message complexity without adversely affecting the average message complexity, detection latency and space complexity. The main idea is as follows. Instead of evaluating the locally stable predicate on a periodic basis, the monitor evaluates the predicate *only after learning* that some process has executed an event. Specifically, on learning that some process in the system has executed an event, the monitor evaluates the predicate by taking two snapshots of the system one after another. The monitor takes the second snapshot only after it has the first snapshot to ensure that the two snapshots form a consistent interval. Intuitively, the first snapshot resets all dirty bits and the second snapshot is used to test if the predicate has become true. We refer to the act of predicate evaluation by taking two snapshots of the system as evaluateLSP. To account for the degenerate case when the predicate is true initially and, therefore, no event may be executed in the system, the monitor evaluates the predicate once in the beginning.

We now provide more details of the algorithm. Processes are arranged in the form of a tree and the root of the tree acts as a monitor. Whenever a process executes an event, it sends a message to the root, via the tree, requesting it to evaluate the predicate since the predicate may have become true. Requests to the root are forwarded in a "delayed" manner. Specifically, once a process has forwarded a request for predicate evaluation to its parent, it does not forward any more requests to the parent until its previous request has been satisfied. A process considers its previous request to be satisfied if it has participated in at least one instance of predicate evaluation *after* sending/forwarding the request to its parent.

Intuitively, the "delayed" forwarding of requests to the parent helps in reducing the average message complexity of the modified algorithm. This is because of the following reasons. While a process is waiting for its previous request to be satisfied, it may execute many events, but it only sends one request to the parent for all such events. Further, while waiting, a non-leaf process in the tree may receive requests from many children, but it only forwards one request to its parent for all such children.

We refer to this modified algorithm as BoundedLSPD. A formal description of the algorithm is given in Figs. 1 and 2. In the algorithm, each process p_i maintains two variables $status_i$ and $pending_i$. The first variable $status_i$ keeps track of the status of a predicate evaluation request that p_i has forwarded to its parent. If $status_i = \text{NOT\_WAITING}$, then all requests forwarded by p_i to its parent have been satisfied. On the other hand, if $status_i \neq \text{NOT\_WAITING}$, then we say that p_i has an *extant* request for predicate evaluation. The second variable $pending_i$ keeps track of whether p_i has a *pending* request for predicate evaluation that it has received either from itself or one

Algorithm for process p_i:

Variables:
  $status_i$: status of the previous request for predicate evaluation, initially NOT_WAITING;
      possible values are:
        NOT_WAITING: all previous requests have been satisfied
        WAITING_TO_START: waiting to participate in an instance of predicate evaluation
        WAITING_TO_END: waiting for the instance of predicate evaluation in progress to end

  $pending_i$: whether there is a pending request for predicate evaluation, initially false;

(A0) Initial action:
  if (p_i is the root) **then**
      // *start an instance of predicate evaluation (that is, evaluateLSP) in*
      // *case the predicate is true initially*
      send a request for taking a snapshot to all children;
  endif;

(A1) On executing an application event:
  $pending_i :=$ true;

(A2) On receiving a request for predicate evaluation from a child:
  $pending_i :=$ true;

(A3) On $(status_i =$ NOT_WAITING$) \land pending_i$ becoming true:
  $status_i :=$ WAITING_TO_START;
  // *the previous request for predicate evaluation have been satisfied and*
  // *there is a pending request for predicate evaluation*
  $pending_i :=$ false;
  if (p_i is a root process) **then**
      // *start a new instance of predicate evaluation (that is, evaluateLSP)*
      send a request for taking a snapshot to all children;
      $status_i =$ WAITING_TO_END;
  else
      send a request for evaluating the predicate to the parent;
  endif;

(A4) On receiving a request to take snapshot from parent:
  if (it is the first snapshot for predicate evaluation) **and**
      ($status_i =$ WAITING_TO_START) **then**
      $status_i :=$ WAITING_TO_END;
  endif;
  if (p_i is a leaf process) **then**
      record local snapshot and reset all dirty bits;
      send the recorded snapshot to the parent;
      if (it is the second snapshot for predicate evaluation) **and**
          ($status_i =$ WAITING_TO_END) **then**
          $status_i =$ NOT_WAITING;
      endif;
  else
      forward the request for taking a snapshot to all children;
  endif;

Fig. 1. The algorithm with bounded message complexity BoundedLSPD for detecting a locally stable predicate.

of its children, but has not forwarded the request to its parent yet. We say that p_i is *latent* if it has a pending request but no extant request for predicate evaluation (that is, action (A3) is enabled). From the algorithm, p_i forwards a pending request for predicate evaluation to its parent as soon as it becomes latent, that is, $pending_i =$ true and $status_i =$ NOT_WAITING.

The safety of BoundedLSPD follows from Theorem 2, Lemmas 6 and 7. We now show that BoundedLSPD is live. The following lemma can be proved by using induction on the distance of a process from the root in the tree.

Lemma 8. *Any extant request for predicate evaluation at a process is satisfied within $O(d)$ time units.*

Suppose the predicate becomes true after executing an event on process p_i at time t. We call a process p_j an *ancestor* of process p_i if it lies on the path from the root to p_i (both inclusive) in the tree. The following lemma can be proved by using induction on the distance of an ancestor of p_i from p_i in the tree.

Lemma 9. *Every ancestor of p_i becomes latent at least once within $O(d)$ time units of t.*

Specifically, the above lemma implies that

Corollary 10. *The root becomes latent at least once within $O(d)$ time units of t.*

Algorithm for process p_i (continued):

```
(A5) On receiving a snapshot collection from a child:
    if (a snapshot collection has been received from all children) then
        record local snapshot and reset all dirty bits;
        if (it is the second snapshot for predicate evaluation) and
            (status_i = WAITING_TO_END) then
            // previous request sent for predicate evaluation has been satisfied
            status_i := NOT_WAITING;
        endif;
        if (p_i is the root process) then
            test whether the predicate holds for the snapshot collected;
            if (the predicate evaluates to false) and
                (it is the first snapshot for predicate evaluation) then
                send a request to take another snapshot to all children;
            endif;
        else
            send the entire snapshot collection to the parent;
        endif;
    endif;
```

Fig. 2. The algorithm with bounded message-complexity BoundedLSPD for detecting a locally stable predicate (continued).

The root, on becoming latent, initiates an instance of predicate evaluation if it still has not detected that the predicate has become true. From Theorem 4, any instance of predicate evaluation initiated after time t terminates with answer "yes". This, in turn, implies that the algorithm BoundedLSPD is live. Further, an instance of predicate evaluation finishes in $O(d)$ time units, which implies that the detection latency of BoundedLSPD is $O(d)$.

The worst-case message complexity of BoundedLSPD is $O(n(m+1))$. This is because, in the worst-case, each event in the system causes the monitor to take two snapshots to evaluate the predicate. As far as average message complexity is concerned, the derivation given in Section 4 for BasicLSPD can be easily adopted for BoundedLSPD. Specifically, the average number of instances of a snapshot algorithm initiated by the monitor, denoted by η_s, is still given by

$$\eta_s \approx \frac{m}{n} \frac{\delta_e}{\delta_s + \delta_w} + 2.$$

It can be verified that a process sends *at most one* request to its parent for every instance of predicate evaluation initiated by the monitor. Note that each instance of predicate evaluation involves taking two snapshots of the system. Therefore, at most $5(n-1)$ control messages are exchanged for every instance of predicate evaluation. As a result the average message complexity of BoundedLSPD is given by

$$\eta_m \approx \frac{5(n-1)}{2} \eta_s \approx \frac{5(n-1)}{2} \left(\frac{m}{n} \frac{\delta_e}{\delta_s + \delta_w} + 2 \right)$$

$$\approx \frac{2.5 \, m \, \delta_e}{2 \, d \, \delta_c + \delta_w} + 5(n-1).$$

Clearly, BoundedLSPD has the same space complexity as BasicLSPD. Therefore, the modification described in this section gives an algorithm with bounded worst-case message complexity without adversely affecting its performance with respect to other metrics.

Note that it is not necessary to evaluate the predicate after executing any application event. Instead, it is sufficient to evaluate the predicate whenever a process executes an application event that has the potential of changing the value of the predicate (from false to true). Such an event is referred to as a *relevant* event [36]. For termination detection, such an event occurs when process changes its state from active to passive. For deadlock detection, such an event occurs when process changes its state from unblocked to blocked.

6. Applications

First, we identify a class of stable predicates that can be detected efficiently using the algorithm for detecting a locally stable predicate. Next, we show that many algorithms for detecting termination and deadlock can be viewed as special cases of our general algorithm for detecting a locally stable predicate. We also instantiate the general algorithm to derive an efficient algorithm for deadlock detection whose performance is comparable with that of existing deadlock detection algorithms.

6.1. Detecting a subclass of stable predicates

The approach described in Section 3 can only be used to detect a locally stable predicate in general. For any other predicate, safety is guaranteed but liveness is not. In this section, we identify a class of stable predicates for which even liveness is ensured.

Consider a stable predicate b that is a function of k locally stable predicates $b_1, b_2, \ldots, b_k$. If b is a monotonic function of all its arguments, then it is indeed possible to use the algorithm BasicLSPD to detect b. Some examples of predicates that are monotonic functions of all their arguments are

1. $b_1 \wedge b_2 \wedge \cdots \wedge b_k$,
2. $b_1 \vee b_2 \vee \cdots \vee b_k$,
3. $(b_1 \vee b_2) \wedge (b_3 \vee b_4) \wedge \cdots \wedge (b_{k-1} \vee b_k)$, when k is even.

Note that, although b is stable, it *may not be* locally stable. For example, suppose $b = b_1 \vee b_2$, where b_1 and b_2 are locally stable predicates. Once one of the disjuncts, say b_1, becomes true, b also becomes true. However, the other disjunct b_2 may still be false. Therefore, values of variables of b_2, and consequently of b, may continue to change implying that b is not locally stable. We assume that false < true. The assignment of values to the arguments of b can be represented by a boolean vector of size k; the jth entry of the boolean vector refers to the value of the locally stable predicate b_j. Let X and Y be two boolean vectors representing possible assignment of values to the arguments of b. We say that $X \leqslant Y$ if

$$\langle \forall j : 1 \leqslant j \leqslant k : X[j] \leqslant Y[j] \rangle.$$

We use $b(X)$ to denote the value of b when evaluated for the boolean vector X. The predicate b is said to be a monotonic function of all its arguments if it satisfies the following:

$$\langle \forall X, Y :: X \leqslant Y \implies b(X) \leqslant b(Y) \rangle$$

which in turn implies the following:

$$\langle \forall X, Y :: (X \leqslant Y) \wedge b(X) \ \Rightarrow \ b(Y) \rangle.$$

We now show that any stable predicate derived from locally stable predicates using $\wedge$ and $\vee$ operators satisfies the monotonicity property.

Lemma 11. *Consider a predicate b derived from k other locally stable predicates $b_1, b_2, \ldots, b_k$ using $\wedge$ and $\vee$ operators. Then b satisfies the monotonicity property.*

Proof. Let X and Y be two boolean vectors representing assignment of values to the arguments of b. Assume that $X \leqslant Y$ and $b(X)$ holds. We have to show that $b(Y)$ holds as well. Without loss of generality, assume that the formula is *fully parenthesized*. The proof is by induction on the length of the formula, say l, where the length of the formula is defined as the number of operators in the formula.

Base case ($l = 0$): Clearly, either b is a constant or $b = b_j$ for some j where $1 \leqslant j \leqslant k$. It can be easily verified that, in either case, $b(Y)$ holds.

Induction step ($l > 0$): Assume that the lemma holds when the length of the formula is at most $l - 1$. We now show that the lemma holds when the length of the formula is l. There are two cases to consider depending on the topmost operator, say op, in the formula:

Case 1 ($op = \wedge$): In this case,

$$b(b_1, b_2, \ldots, b_k) = u(b_1, b_2, \ldots, b_k) \wedge v(b_1, b_2, \ldots, b_k).$$

Therefore, we have,

$$(X \leqslant Y) \wedge b(X)$$

$$\equiv \{ \ b = u \wedge v \ \}$$

$$(X \leqslant Y) \wedge u(X) \wedge v(X)$$

$$\equiv \{ \text{ predicate calculus } \}$$

$$\Big((X \leqslant Y) \wedge u(X) \Big) \bigwedge \Big((X \leqslant Y) \wedge v(X) \Big)$$

$$\Rightarrow \{ \text{ using induction hypothesis } \}$$

$$u(Y) \wedge v(Y)$$

$$\equiv \{ \ b = u \wedge v \ \}$$

$$b(Y).$$

Therefore, in this case, b satisfies the monotonicity property.
Case 2 ($op = \vee$): The proof for the second case is similar to that for the first case.
This establishes the lemma. □

For convenience, we refer to stable predicates that can be expressed as monotonic functions of other locally stable predicates as *monotonically decomposable stable predicates*.

A simple approach for detecting b is to use an instance of BasicLSPD for each locally stable predicate in b. However, instead of taking a separate snapshot for each locally stable predicate, the monitor can take only one snapshot and use it to evaluate those locally stable predicates in b that have not become true so far. Based on the truth values of the locally stable predicates in b, the truth value of b itself can be computed. We refer to this algorithm as BasicMSPD. As in the case of BasicLSPD, BasicMSPD also has unbounded message complexity in the worst-case. Using ideas presented in Section 5, we can similarly obtain an algorithm BoundedMSPD for detecting b that has bounded message complexity. Clearly, BoundedMSPD has same message complexity (worst-case and average-case) and detection latency as BoundedLSPD.

6.2. Detecting termination of a distributed computation

The termination detection problem involves detecting when an ongoing distributed computation has ceased all its activities. The distributed computation satisfies the following rules. A process can be either in an *active* state or a *passive* state. A process can send a message only when it is active. An active process can become passive at any time. A passive process becomes active on receiving a message. The computation is said to have *terminated* when all processes have become passive and all channels have become empty. (The termination detection problem arises when a distributed computation terminates *implicitly* and therefore a separate algorithm is required to detect the termination [48]. Examples of such computations can be found in [48].)

A large number of algorithms that have been developed for termination detection can be viewed as special cases of our approach (e.g., [41,37,14,40,19,13,24]). To detect termination, we need to test for two conditions. First, all processes have become passive. Second, all channels have become empty. Passiveness of a process is a local property and can be tested quite easily by the process. However, emptiness of a channel depends on the states of two processes connected by the channel and is therefore not a local property. Typically, three approaches have been used to test for the emptiness of channels: *acknowledgment-based*, *message counting-based* and *channel flushing-based*.

6.2.1. Acknowledged-based approach

In this approach, when a process receives an application message, it sends an acknowledgment message to the source process. Therefore, once a process has received an acknowledgment message for every application message it has sent so far, it knows that all its outgoing channels are empty. For a process p_i, let $passive_i$ be true if the process is passive and false otherwise. Also, let $missing_i$ denote the difference between the number of application messages that have been sent by p_i and the number of acknowledgment messages that have been received by p_i. The termination condition can be formulated as

$$\langle \forall i : 1 \leqslant i \leqslant n : passive_i \wedge (missing_i = 0) \rangle.$$

Hélary and Raynal [19] use the above formulation of the termination condition to detect termination of a distributed

computation. In their approach, a process takes a snapshot in a lazy manner, that is, it delays recording of its local snapshot until it has become passive and all its application messages have been acknowledged. However, instead of using dirty bits, each process p_i maintains a flag cp_i that is true if and only if p_i continuously stayed passive since cp_i was last reset. The flag cp_i acts as a dirty bit for the variable $passive_i$. There is no need to maintain a dirty bit for the other variable $missing_i$ because the rules of the computation ensure that the dirty bit for the variable $passive_i$ is set if and only if the dirty bit for the variable $missing_i$ is set.

6.2.2. Message counting-based approach

In this approach, each process p_i maintains a *deficit counter*, denoted by $deficit_i$, that tracks the difference between the number of messages it has sent and the number of messages it has received. The termination condition can be formulated as

$$\langle \forall i : 1 \leqslant i \leqslant n : passive_i \rangle \ \wedge \ \left(\sum_{i=1}^{n} deficit_i = 0 \right).$$

The termination detection algorithm by Safra [14] uses the above formulation of the termination condition. Again, instead of using dirty bits, each process p_i is associated with a color, denoted by $color_i$, which is white if p_i stayed continuously passive since $color_i$ was last reset; otherwise it is black. Some other examples of termination detection algorithms that use message counting and are special cases of our algorithm include Mattern's four counter algorithm [37], Mattern's sticky-flag algorithm [40] and Sinha *et al.*'s two phase algorithm [46].

6.2.3. Channel flushing-based approach

This approach assumes that all channels satisfy the FIFO property. The emptiness of a channel is tested by sending a marker message along that channel. If the process at the receiving end of the channel receives the marker message without receiving any application message before it, then the channel was empty; otherwise it was not. A snapshot of the system is collected using two waves [49]. The first wave collects the local state of each process. A process, on recording its local state, sends a marker message along all its outgoing channels. The second wave collects the state of each channel. On receiving the second wave, a process waits until it has received a marker message along every incoming channel before propagating the wave further. Note that the two waves of a snapshot are not required to form a consistent interval and therefore can overlap. If a tree-based approach is used to record a snapshot of the system, then process states can be recorded in the broadcast phase and channel states can be recorded in the convergecast phase.

It is possible to evaluate the termination condition using only a single wave per snapshot. Specifically, each wave collects local states of all processes with respect to the *current* snapshot and states of all channels with respect to the *previous* snapshot. Let $allPassive$ denote the predicate that all processes are passive, and $allEmpty$ denote the predicate that all channels are empty. Then the termination condition can be represented as

$$termination \ \triangleq \ allPassive \wedge allEmpty.$$

Let C and D be two consecutive snapshots such that $[C, D]$ is quiescent with respect to $termination$. We have,

$$termination(D)$$

$\equiv$ { definition of $termination$ }

$\quad all\ Passive(D) \wedge allEmpty(D)$

$\equiv$ { $[C, D]$ is quiescent with respect to $termination$ }

$\quad all\ Passive(D) \wedge allEmpty(D) \wedge$

$\quad (allEmpty(C) = allEmpty(D))$

$\Rightarrow$ { predicate calculus }

$\quad all\ Passive(D) \wedge allEmpty(C).$

In other words, to evaluate the termination condition with respect to the snapshot D, as far as a channel is concerned, it is safe to use its state with respect to the earlier snapshot C. Termination detection algorithms based on using marker messages to detect emptiness of channels can be found in [41,11]. For example, the quiescence detection algorithm in [11] uses the ring-based approach to record a snapshot. Moreover, a snapshot session is aborted as soon as it is detected that the computation has not terminated or the interval is not quiescent. The process that aborts the snapshot session then starts a new snapshot session and forwards the token to the next process in the ring.

When channel flushing-based approach is used to detect emptiness of a channel, the message complexity is no longer $O(n(m + 1))$ but increases to $O(c(m + 1))$, where c is the number of channels in the communication topology.

6.3. Detecting deadlock in a distributed system

Deadlock may occur when processes require access to multiple resources at the same time. Ho and Ramamoorthy [20] present a two-phase algorithm to detect a deadlock in a distributed database system under the AND request model. In Ho and Ramamoorthy's two-phase algorithm [20], a wait-for graph is constructed in each phase by collecting a snapshot of the system in the form of *process status tables*. A deadlock is announced if the cycle observed in the first-phase is also detected in the second phase. Two phases are used because the snapshot collected in a phase may be inconsistent, and, therefore, a cycle detected in the wait-for graph constructed using such a snapshot may not actually exist in the system. The second phase is used to validate the cycle observed in the first phase. The algorithm is flawed because it does not correctly test for quiescence of the interval between the first phase and second phase before evaluating the deadlock condition. Specifically, their quiescence detection approach works only if status tables maintain information about transactions (and not processes) and transactions follow the two-phase locking discipline. Although the two-phase deadlock detection algorithm as described in [20] is incorrect, it can be fixed using the ideas given in this paper as discussed next. A correct version of the two phase deadlock detection algorithm can also be found in [26].

We briefly explain how our general approach can be instantiated to detect a deadlock under any request model (and also

fixes Ho and Ramamoorthy's two-phase deadlock detection algorithm). To detect a deadlock, each process maintains a table that tracks the current status of the process: there is an entry in the table for each resource that a process is either holding or waiting on. Moreover, there is a dirty bit associated with each entry in the table. Whenever an entry is added to the table, the dirty bit is initially set to its unclean state. When a process takes a local snapshot, it only records those entries in the table whose dirty bit is in the clean state. As discussed in Section 3.2.2, immediately after taking the snapshot, all dirty bits are reset to their clean states. Note that a snapshot of the system only collects those entries that were in the table when the last snapshot was taken and have not changed since then. Since more than one process may initiate the snapshot algorithm, for every entry in the table, a separate dirty bit has to be maintained for each possible initiator. The initiator of the snapshot algorithm can then use the entries recorded as part of the snapshot to construct a wait-for graph. (Alternatively, instead of dirty bits, it is possible to use a local clock to timestamp entries in the table.) Consider a wait-for graph G constructed by a process as described above. Given a consistent cut C, let $WFG(C)$ denote the wait-for graph that exists among processes for the cut C. Using properties of interval consistency and interval quiescence, it can be deduced that

$$\langle \exists \, C : C \text{ is a consistent cut} : G \text{ is a subgraph of } WFG(C) \rangle.$$

In other words, various edges in G are actually *consistent* with each other and, therefore, G can be safely analyzed to obtain meaningful results. Now, depending on the request model, any algorithm for analyzing a wait-for graph can be used to detect a deadlock [21,12,33]. For example, under the AND request model, the wait-for graph should contain a cycle, whereas, under the OR request model, the wait-for graph should contain a knot.

6.3.1. Computing maximum deadlocked set

Our deadlock detection algorithm can be used to determine the subset of all processes currently involved in a deadlock, which is referred to as *maximum deadlocked set* [8]. The algorithm for computing the maximum deadlocked set has $O(n)$ message complexity and $O(d)$ time complexity, where n is the number of processes in the system and d is the diameter of the communication topology.

6.3.2. Improving performance of the deadlock detection algorithm

Note that the message complexity of the deadlock detection algorithm described above depends on the total number of processes in the system. However, it can be modified to obtain a more efficient deadlock detection algorithm whose message complexity depends on the size of the wait-for graph. Such an algorithm is useful, for example, when most deadlocks involve only a small subset of processes. For convenience, we assume that the communication topology is fully connected.

The main idea is as follows. When a process suspects that it may be involved in a deadlock, instead of taking a snapshot of the entire system, it takes a snapshot of only those processes that can be involved in a deadlock with it. To that end, it first computes its *reachability set*. The reachability set of a process consists of all those processes that can be reached from it through edges in the wait-for graph. To compute the reachability set, we assume that each process knows its *dependency set*, that is, the set of processes that are holding resources that it is currently waiting on. This assumption is also made by most deadlock detection algorithms (e.g., [7,27,12,6,29,35,33]). One way to satisfy this assumption is by treating resources as processes and also including them when constructing the wait-for graph [28]. A process can compute its reachability set in an iterative manner [12]; in the jth iteration, it discovers processes at a distance of j hops from it in the wait-for graph. Let r denote the number of processes in the reachability set and let t denote the diameter of the wait-for graph. Clearly, a process can compute its reachability set using at most $2r$ messages and in at most $2t$ time units. As part of computing its reachability set, the process can also collect a snapshot of all processes in the set. After a process has computed its reachability set, the message complexity of taking the second snapshot is only $2r$ messages and its time complexity is only 2. Therefore, the modified deadlock detection algorithm has overall message complexity of $4r$ and detection latency of $2t + 2$. This compares quite favorably with the existing deadlock detection algorithms. For example, the deadlock detection algorithm by Chen et al. [12] has message complexity of $2r$ and detection latency of $2t$. Their algorithm uses logical timestamps and therefore requires application messages to be modified to carry the timestamps. The deadlock detection algorithm by Lee [33] has low detection latency of $t + 2$ but high message complexity of $2s$, where s is the number of edges in the wait-for graph among processes in the reachability set. The algorithm by Chen et al. [12], to our knowledge, has the best message complexity among all deadlock detection algorithms, and our modified algorithm uses at most twice the number of messages as their algorithm.

7. Achieving fault tolerance

In this section, we describe modifications to our approach for detecting a locally stable predicate to make it fault-tolerant. We assume that processes are unreliable and may fail by crashing. Once a process crashes, it stops executing any events. Further, a crashed process never recovers. A process that never crashes is said to be *correct*; a process that is not correct is said to be *faulty*. We call a process that has not crashed so far as *operational* or *live*. We assume that process crashes do not partition the system. Although processes are unreliable, we assume that channels are reliable in the sense that a message sent by a correct process to another correct process is eventually delivered.

To detect a locally stable predicate in the presence of process crashes, in general, it is necessary that the crash of a process can be *reliably* detected by its neighboring processes. This is because the same assumption is necessary to solve the

termination detection problem [42], and detecting termination is a special case of detecting a locally stable predicate.

Once a process crashes, its state (specifically, its state just before the crash) becomes inaccessible to other processes. Therefore, we assume that evaluating the predicate only involves *currently operational processes and channels that are incident on them*. Note that a faulty process before crashing may have sent one or more messages to other processes in the system, which may still be in transit at the time of the crash. Such messages have to be somehow "taken into account" before announcing that the predicate has become true; otherwise the safety property may be violated. For instance, consider the termination detection problem and assume that all operational processes are passive and all channels between them are empty. However, termination should not be announced yet as a message from a crashed process may make an operational process, which is currently passive, active. There are two approaches to deal with this problem. The first approach is to assume the existence of special primitive, such as *return-flush* or *fail-flush*, that allows an operational process to flush its incoming channel with a crashed process [51,30]. The second approach is to assume that an operational process, on detecting crash of its neighbor, simply *freezes* its incoming channel with the crashed process and does not accept any more messages received along that channel [50,42]. For the ease of exposition, we take the second approach. However, the ideas in this section can be used with the first approach as well.

With the channel freezing approach, it is sufficient to evaluate the predicate on the currently operational processes and the channels *between* them. Therefore, it is sufficient to collect local states of currently operational processes only. Note that, in the presence of process crashes, the safety of an algorithm specifically designed for a failure-free environment (such as BasicLSPD or BoundedLSPD) is not violated; only its liveness may be violated [42]. To ensure liveness, the main problems that need to be addressed are as follows. First, on occurrence of a failure, a new monitor may have to be elected because the old monitor may have crashed. Second, a new spanning tree has to be constructed on the set of currently operational processes because the failure may have disconnected the current spanning tree. To solve the two problems, we can use the approach described in [42]. We describe it briefly here for the sake of completeness. Whenever a process crashes, we reconstruct the spanning tree using the tree construction algorithm proposed by Awerbuch [5]. An advantage of Awerbuch's algorithm is that different processes can start the algorithm at different times. So, whenever a process learns about a new failure, it simply starts a *new instance* of the spanning tree construction algorithm. (Any old instance of the tree construction algorithm is aborted.) We differentiate between various instances of the spanning tree construction algorithm by using the process' knowledge about the failure of other processes when it starts the new instance, which we refer to as *instance identifier*.

A process may learn about the failure of a process either directly via its failure detector or through the instance identifier of a message received. Note that the former can only provide information about the failure of a neighboring process, whereas the latter can provide information about the failure of any process. In any case, on learning about a new failure, a process starts a new instance of the spanning tree construction algorithm as explained earlier.

If no more failures occur for a sufficiently long period of time, then all operational processes eventually learn about the failure of all crashed processes. Therefore, eventually, all operational processes start the same instance of the spanning tree construction algorithm and a valid spanning tree is eventually constructed. Once the tree construction algorithm terminates, the root of the tree elects itself as the monitor. Thereafter, it uses the new tree to collect a snapshot of currently operational processes until another process fails.

Let f denote the number of processes that crash during an execution. Also, let c denote the number of channels in the communication topology. It can be shown that the number of additional control messages exchanged due to process crashes is given by $O(f(c+n \log n))$ [42]. This implies that the cost of handling process crashes increases in proportion to the number of processes that actually crash during an execution.

8. Related work

The work most relevant to our work are algorithms for detecting a locally stable predicate and those for detecting a stable predicate.

Marzullo and Sabel [36] describe an approach for detecting a locally stable predicate using the notion of *weak vector clock*. A weak vector clock, unlike the Fidge/Mattern's vector clock [39,16], is updated only when an event that is relevant with respect to the predicate is executed. Whenever a process sends a message, it timestamps the message with the current value of its local (weak) vector clock. A process, on receiving a message, updates its weak vector clock by taking the component-wise maximum of its own clock and the timestamp of the message. (Of course, if the receive event of the message is a relevant event, then it also increments its entry in the weak vector clock.) Thus Marzullo and Sabel's approach requires application messages to be modified to carry a vector timestamp of size n, where n is the number of processes in the system. This makes their approach unscalable as the number of processes in the system increases. Further, in their approach, local snapshot of each process is assigned a vector timestamp of size n, which is used at the time of evaluating the predicate. As a result, their space complexity is $O(n(n+s))$, which is higher than that our approach in case $s = o(n)$. (Recall that $O(s)$ is the amount of space required to store local snapshot of a single process.)

Based on their general approach, Marzullo and Sabel present two different algorithms for detecting a locally stable predicate. Similar to BasicLSPD, the first algorithm has unbounded message complexity in the worst-case. The second algorithm, however, has worst-case message complexity of $O(m d)$. Basically, in the second algorithm, a process sends its local snapshot to a monitor whenever it executes a relevant event. If a message on average travels a distance of $d/2$, the average message complexity of the second algorithm is approximately $m d/2$.

Observe that the algorithm for detecting a locally stable predicate given by Marzullo and Sabel [36] can also be used to detect a monotonically decomposable stable predicate described in Section 6. However, with their solution, each application message may have to carry up to k vectors, each of size n, where k is the number of locally stable predicates on which the predicate depends. This is because the set of relevant events may be different for different locally stable predicates and, as a result, a different weak vector clock may have to be maintained for each locally stable predicate. This results in high application message overhead of $O(n\,k)$. On the other hand, in our algorithm, application messages do not carry any control information and, therefore, application message overhead due to detection algorithm is zero.

A locally stable predicate is also a stable predicate. Therefore, an algorithm for detecting a stable predicate can also be used for detecting a locally stable predicate. The only approach that we know of for detecting a stable predicate is by repeatedly taking *consistent* snapshots of the system until the predicate evaluates to true. Taking a consistent snapshot of the system requires different processes to coordinate their actions to ensure that every pair of local snapshots are mutually concurrent. As a result, any algorithm for taking a consistent snapshot of the system, has to constrain the process as to when it should record its local state. On the other hand, when a snapshot is not required to be consistent, processes are free to record their local states *opportunistically*. For example, it may be better for a process to record its local state when it is idle or waiting for an I/O operation.

Lai and Yang [31] define the notion of *strongly stable predicate*. Intuitively, a stable predicate is strongly stable if it can be evaluated correctly even for an inconsistent cut. Formally,

Definition 6 (*strongly stable predicate*). A stable predicate b is strongly stable if for all cuts C and D (consistent as well as inconsistent),

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

The class of strongly stable predicates is incomparable with the class of locally stable predicates. Consider the following predicate:

$$b_{\text{sum}} \triangleq x_1 + x_2 + \cdots + x_n \geqslant s,$$

where each x_i is a monotonically non-decreasing variable on process p_i and s is a constant. Clearly, b_{sum} is not a locally stable predicate because the values of x_i's may continue to increase even after b_{sum} has become true. It can be easily verified that b_{sum} is a strongly stable predicate.

Note that, for a stable predicate to be strongly stable, if it evaluates to true for some consistent cut C, then it is not sufficient for the predicate to evaluate to true for every inconsistent cut D that follows C. It should be the case that if the predicate evaluates to true for some inconsistent cut C, then it should also evaluate to true for every consistent cut D that follows C. The former only guarantees liveness of a detection algorithm. The latter is necessary to ensure safety of the detection algo-

rithm. To show that the class of locally stable predicates is not a subset of the class of strongly stable predicates, consider the following formulation of termination based on deficit counters:

$$\langle \forall i : 1 \leqslant i \leqslant n : passive_i \rangle \wedge \left(\sum_{i=1}^{n} deficit_i = 0 \right).$$

Evidently, the predicate is locally stable. However, it is not strongly stable because it may evaluate to true for an inconsistent cut even though the system has not terminated.

Similar to the notion of locally stable property, we can define the notion of strongly stable property as follows: a stable property is strongly stable if there exists at least one way to formulate the property such that the formulation corresponds to a strongly stable predicate. An interesting question that still remains is: *how does the class of locally stable "properties" compare with the class of strongly stable "properties"?*

9. Conclusion and future work

In this paper, we have described 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 need not be modified to carry any control information. It also unifies several known algorithms for detecting two important locally stable properties, namely termination and deadlock. We have also shown that any stable predicate that can be expressed as a monotonic function of other locally stable predicates can be efficiently detected using our approach. Finally, we have described extensions to our approach to make it tolerant to process crashes.

In this paper, we assume 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, the 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 a future work, we plan to investigate efficient ways for monitoring an application program in a transparent manner.

References

[1] A. Acharya, B.R. Badrinath, Recording distributed snapshots based on causal order of message delivery, Inform. Process. Lett. (IPL) 44 (6) (1992) 317–321.

[2] S. Alagar, S. Venkatesan, An optimal algorithm for recording snapshots using casual message delivery, Inform. Process. Lett. (IPL) 50 (1994) 311–316.

[3] R. Atreya, N. Mittal, V.K. Garg, Detecting locally stable predicates without modifying application messages, in: Proceedings of the seventh International Conference on Principles of Distributed Systems (OPODIS), La Martinique, France, 2003, pp. 20–33.

[4] H. Attiya, J. Welch, Distributed Computing: Fundamentals, Simulations and Advanced Topics, second ed., Wiley, New York, 2004.

[5] B. Awerbuch, Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems, in: Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC), ACM Press, New York, NY, United States, 1987, pp. 230–240.

[6] A. Boukerche, C. Tropper, A distributed graph algorithm for the detection of local cycles and knots, IEEE Trans. Parallel Distributed Systems (TPDS) 9 (8) (1998) 748–757.

[7] G. Bracha, S. Toueg, Distributed deadlock detection, Distributed Comput. (DC) 2 (3) (1987) 127–138.

[8] J. Brzezinski, J.M. Hélary, M. Raynal, M. Singhal, Deadlock models and a general algorithm for distributed deadlock detection, J. Parallel Distributed Comput. (JPDC) 31 (2) (1995) 112–125.

[9] S. Chandrasekaran, S. Venkatesan, A message-optimal algorithm for distributed termination detection, J. Parallel Distributed Comput. (JPDC) 8 (3) (1990) 245–252.

[10] K.M. Chandy, L. Lamport, Distributed snapshots: determining global states of distributed systems, ACM Trans. Comput. Systems 3 (1) (1985) 63–75.

[11] K.M. Chandy, J. Misra, An example of stepwise refinement of distributed programs: quiescence detection, ACM Trans. Programm. Languages Systems (TOPLAS) 8 (3) (1986) 326–343.

[12] S. Chen, Y. Deng, P.C. Attie, W. Sun, Optimal deadlock detection in distributed systems based on locally constructed wait-for graphs, in: Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS), May 1996, pp. 613–619.

[13] M. Demirbas, A. Arora, An optimal termination detection algorithm for rings, Technical Report OSU-CISRC-2/00-TR05, The Ohio State University, February 2000.

[14] E.W. Dijkstra, Shmuel Safra's Version of Termination Detection, EWD Manuscript 998, available at ⟨http://www.cs.utexas.edu/users/EWD/⟩, 1987.

[15] E.W. Dijkstra, C.S. Scholten, Termination detection for diffusing computations, Inform. Process. Lett. (IPL) 11 (1) (1980) 1–4.

[16] C.J. Fidge, Logical time in distributed computing systems, IEEE Computer 24 (8) (1991) 28–33.

[17] E. Fromentin, M. Raynal, Inevitable global states: a concept to detect unstable properties of distributed computations in an observer independent way, in: Proceedings of the sixth IEEE Symposium on Parallel and Distributed Processing (SPDP), 1994, pp. 242–248.

[18] J.-M. Hélary, C. Jard, N. Plouzeau, M. Raynal, Detection of stable properties in distributed applications, in: Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC), 1987, pp. 125–136.

[19] J.-M. Hélary, M. Raynal, Towards the construction of distributed detection programs, with an application to distributed termination, Distributed Comput. (DC) 7 (3) (1994) 137–147.

[20] G.S. Ho, C.V. Ramamoorthy, Protocols for deadlock detection in distributed database systems, IEEE Trans. Software Eng. 8 (6) (1982) 554–557.

[21] R.C. Holt, Some deadlock properties of computer systems, ACM Comput. Surveys (CSUR) 4 (1972) 179–195.

[22] S.-T. Huang, Detecting termination of distributed computations by external agents, in: Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS), 1989, pp. 79–84.

[23] J.R. Jagannathan, R. Vasudevan, Comments on protocols for deadlock detection in distributed database systems, IEEE Trans. Software Eng. 9 (3) (1983) 371.

[24] A.A. Khokhar, S.E. Hambrusch, E. Kocalar, Termination detection in data-driven parallel computations/applications, J. Parallel Distributed Comput. (JPDC) 63 (3) (2003) 312–326.

[25] E. Knapp, Deadlock detection in distributed databases, ACM Comput. Surveys (CSUR) 19 (4) (1987) 303–328.

[26] A.D. Kshemkalyani, M. Singhal, Correct two-phase and one-phase deadlock detection algorithms for distributed systems, in: Proceedings of the IEEE Symposium on Parallel and Distributed Processing (SPDP), December 1990, pp. 126–129.

[27] A.D. Kshemkalyani, M. Singhal, Efficient detection and resolution of generalized distributed deadlocks, IEEE Trans. Software Eng. 20 (1) (1994) 43–54.

[28] A.D. Kshemkalyani, M. Singhal, On characterization and correctness of distributed deadlock detection, J. Parallel Distributed Comput. (JPDC) 22 (1) (1994) 44–59.

[29] A.D. Kshemkalyani, M. Singhal, A one-phase algorithm to detect distributed deadlocks in replicated databases, IEEE Trans. Knowledge Data Eng. 11 (6) (1999) 880–895.

[30] T.-H. Lai, L.-F. Wu, An $(N − 1)$-resilient algorithm for distributed termination detection, IEEE Trans. Parallel Distributed Systems (TPDS) 6 (1) (1995) 63–78.

[31] T.-H. Lai, T.H. Yang, On distributed snapshots, Inform. Process. Lett. (IPL) 25 (3) (1987) 153–158.

[32] L. Lamport, Time clocks and the ordering of events in a distributed system, Comm. ACM (CACM) 21 (7) (1978) 558–565.

[33] S. Lee, Fast, centralized detection and resolution of distributed deadlocks in the generalized model, IEEE Trans. Software Eng. 30 (9) (2004) 561–573.

[34] N.R. Mahapatra, S. Dutt, An efficient delay-optimal distributed termination detection algorithm, Department of Computer Science and Engineering, University of Buffalo. The State University of New York, November 2001, Technical Report No. 2001-16.

[35] D. Manivannan, M. Singhal, An efficient distributed algorithm for detecting knots and cycles in a distributed graph, IEEE Trans. Parallel Distributed Systems (TPDS) 14 (2003) 961–972.

[36] K. Marzullo, L. Sabel, Efficient detection of a class of stable properties, Distributed Comput. (DC) 8 (2) (1994) 81–91.

[37] F. Mattern, Algorithms for distributed termination detection, Distributed Comput. (DC) 2 (3) (1987) 161–175.

[38] F. Mattern, Global quiescence detection based on credit distribution and recovery, Inform. Process. Lett. (IPL) 30 (4) (1989) 195–200.

[39] F. Mattern, Virtual time and global states of distributed systems, in: Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG), Elesevier Science Publishers B.V., North-Holland, 1989, pp. 215–226.

[40] F. Mattern, H. Mehl, A. Schoone, G. Tel, Global virtual time approximation with distributed termination detection algorithms, Technical Report RUU-CS-91-32, University of Utrecht, The Netherlands, 1991.

[41] J. Misra, Detecting termination of distributed computations using markers, in: Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC), 1983, pp. 290–294.

[42] N. Mittal, F.C. Freiling, S. Venkatesan, L.D. Penso, Efficient reductions for wait-free termination detection in crash-prone systems, Technical Report AIB-2005-12, Department of Computer Science, Rheinisch-Westfälische Technische Hochschule (RWTH), Aachen, Germany, June 2005.

[43] N. Mittal, S. Venkatesan, S. Peri, Message-optimal and latency-optimal termination detection algorithms for arbitrary topologies, in: Proceedings of the 18th Symposium on Distributed Computing (DISC), Amsterdam, The Netherlands, 2004, pp. 290–304.

[44] S. Peri, N. Mittal, On termination detection in an asynchronous distributed system, in: Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems (PDCS), San Francisco, California, USA, September 2004, pp. 209–215.

[45] M. Singhal, Deadlock detection in distributed systems, IEEE Comput. 22 (1989) 37–48.

[46] A. Sinha, L. Kalé, B. Ramkumar, A dynamic and adaptive quiescence detection algorithm, Technical Report 93-11, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 1993.

[47] G. Stupp, Stateless termination detection, in: Proceedings of the 16th Symposium on Distributed Computing (DISC), Toulouse, France, 2002, pp. 163–172.

[48] G. Tel, Distributed control for AI, Technical Report UU-CS-1998-17, Information and Computing Sciences, Utrecht University, The Netherlands, 1998.

[49] G. Tel, Introduction to Distributed Algorithms, second ed., Cambridge University Press (US Server), Cambridge, 2000.

[50] Y.-C. Tseng, Detecting termination by weight-throwing in a faulty distributed system, J. Parallel Distributed Comput. (JPDC) 25 (1) (1995) 7–15.

[51] S. Venkatesan, Reliable protocols for distributed termination detection, IEEE Trans. Reliability 38 (1) (1989) 103–110.
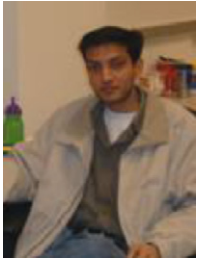
Ranganath Atreya received his B.E. degree in mechanical engineering with distinction from Bangalore University, India in 2000 and his M.S. degree in computer science from The University of Texas at Dallas in 2004. He was the recipient of the National Award for the best B.E. project in 2000. His research interests include computer networking and distributed computing. He has been working at Amazon.com, Inc. since January 2005.

Neeraj Mittal received his B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Delhi in 1995 and M.S. and Ph.D. degrees in computer science from the University of Texas at Austin in 1997 and 2002, respectively. He is currently an assistant professor in the Department of Computer Science and a co-director of the Advanced Networking and Dependable Systems Laboratory (ANDES) at the University of Texas at Dallas. His research interests include distributed systems, mobile computing, networking and databases.

Ajay D. Kshemkalyani received the Ph.D. degree in computer and information science from Ohio State University in 1991 and the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1987. His research interests are in computer networks, distributed computing, algorithms, and concurrent systems. He has been an associate professor at the University of Illinois at Chicago since 2000, before which he spent several years at IBM Research Triangle Park working on various aspects of computer networks. He is a member of the ACM and a senior member of the IEEE and the IEEE Computer Society. In 1999, he received the US National Science Foundation's CAREER Award.

Vijay K. Garg received his B.Tech. degree in computer science from the Indian Institute of Technology, Kanpur in 1984 and M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California at Berkeley in 1985 and 1988, respectively. He is currently a full professor in the Department of Electrical and Computer Engineering and the director of the Parallel and Distributed Systems Laboratory at the University of Texas, Austin. His research interests are in the areas of distributed systems and discrete event systems. He is the author of the books *Elements of Distributed Computing* (Wiley & Sons, 2002), *Principles of Distributed Systems* (Kluwer, 1996) and a co-author of the book *Modeling and Control of Logical Discrete Event Systems* (Kluwer, 1995).

Mukesh Singhal is a Full Professor and Gartener Group Endowed Chair in Network Engineering in the Department of Computer Science at The University of Kentucky, Lexington. From 1986 to 2001, he was a faculty in Computer and Information Science at The Ohio State University. He received a Bachelor of Engineering degree in Electronics and Communication Engineering with high distinction from Indian Institute of Technology, Roorkee, India, in 1980 and a Ph.D. degree in Computer Science from University of Maryland, College Park, in May 1986. His current research interests include distributed systems, wireless and mobile computing systems, computer networks, computer security, and performance evaluation. He has published over 180 refereed articles in these areas. He has coauthored three books titled *Advanced Concepts in Operating Systems*, McGraw-Hill, New York, 1994, *Data and Computer Communications: Networking and Internetworking*, CRC Press, 2001, and *Readings in Distributed Computing Systems*, IEEE Computer Society Press, 1993. He is a Fellow of IEEE. He is a recipient of 2003 IEEE Technical Achievement Award. He is currently serving in the editorial board of *IEEE Transactions on Parallel and Distributed Systems* and *IEEE Transactions on Computers*. From 1998 to 2001, he served as the Program Director of Operating Systems and Compilers program at National Science Foundation.