Monitoring Stable Properties in Dynamic Peer-to-Peer Distributed Systems

Sathya Peri and Neeraj Mittal

Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083, USA
sathya.p@student.utdallas.edu neerajm@utdallas.edu

Abstract. Monitoring a distributed system to detect a stable property is an important problem with many applications. The problem is especially challenging for a *dynamic* distributed system because the set of processes in the system may change with time. In this paper, we present an efficient algorithm to determine whether a stable property has become true in a system in which processes can join and depart the system at any time. Our algorithm is based on maintaining a spanning tree of processes that are currently part of the system. The spanning tree, which is dynamically changing, is used to periodically collect local states of processes such that: (1) all local states in the collection are *consistent* with each other, and (2) the collection is *complete*, that is, it contains all local states that are necessary to evaluate the property and derive meaningful inferences about the system state.
Unlike existing algorithms for stable property detection in a dynamic environment, our algorithm is general in the sense that it can be used to evaluate *any* stable property. Further, it does not assume the existence of any permanent process. Processes can join and leave the system while the snapshot algorithm is in progress.

1 Introduction

One of the fundamental problems in distributed systems is to detect whether some stable property has become true in an ongoing distributed computation. A property is said to be stable if it stays true once it becomes true. Some examples of stable properties include "system is in terminated state", "a subset of processes are involved in a circular wait" and "an object is a garbage". The stable property detection problem has been well-studied and numerous solutions have been proposed for solving the general problem (*e.g.*, [1–3]) as well as its special cases (*e.g.*, [4–10]). However, most of the solutions assume that the system is *static*, that is, the set of processes is fixed and does not change with time.

With the advent of new computing paradigms such as grid computing and peer-to-peer computing, *dynamic* distributed systems are becoming increasingly popular. In a dynamic distributed system, processes can join and leave the ongoing computation at anytime. Consequently, the set of processes in the system may change with time. Dynamic distributed systems are especially useful for solving *large-scale problems* that require vast computational power. For example, distributed.net [11] has undertaken several projects that involve searching a large state-space to locate a solution. Some examples of such projects include

RC5-72 to determine a 72-bit secret key for the RC5 algorithm, and OGR-25 to compute the Optimal Golomb Ruler with 25 and more marks.

Although several algorithms have been proposed to solve the stable property detection problem in a dynamic environment, they suffer from one or more of the following limitations. First, to the best of our knowledge, all existing algorithms solve the detection problem for special cases such as property is either termination [12–14] or can be expressed as conjunction of local predicates [15]. Second, most of the algorithms assume the existence of permanent processes that never leave the system [13, 15, 14]. Third, some of the algorithms assume that processes can join but cannot leave the system until the detection algorithm has terminated [12, 14]. Fourth, the algorithm by Darling and Mayo [15] assumes that processes are equipped with local clocks that are weakly synchronized.

In this paper, we describe an algorithm to detect a stable property for a dynamic distributed system that does not suffer from any of the limitations described above. Our approach is based on maintaining a spanning tree of all processes currently participating in the computation. The spanning tree, which is dynamically changing, is used to collect local snapshots of processes periodically. Processes can join and leave the system while a snapshot algorithm is in progress. We identify sufficient conditions under which a collection of local snapshots can be safely used to evaluate a stable property. Specifically, the collection has to be consistent (local states in the collection are pair-wise consistent) and *complete* (no local state necessary for correctly evaluating the property is missing from the collection). We also identify a condition that allows the current root of the spanning tree to detect termination of the snapshot algorithm even if the algorithm was initiated by an "earlier" root that has since left the system. Due to lack of space, formal description of our algorithm and proofs of various lemmas and theorems have been omitted and can be found in [16].

2 System Model and Notation

2.1 System Model

We assume an asynchronous distributed system in which processes communicate with each other by exchanging messages. There is no global clock or shared memory. Processes can join and leave the system at any time. We do not assume the existence of any permanent process. We, however, assume that there is at least one process in the system at any time and processes are reliable. For ease of exposition, we assume that a process can join the system at most once. If some process wants to join the system again, it joins it as a different process. This can be ensured by using incarnation numbers.

When a process sends a message to another process, we say that the former process has an *outgoing channel* to the latter process. Alternatively, the latter process has an *incoming channel* to the former process. We make the following assumptions about channels. First, any message sent to a process that never leaves the system is eventually delivered. This holds even if the sender of the message leaves the system after sending the message but before the message is delivered. Second, any message sent by a process that never leaves the system to a process that leaves the system before the message is delivered is eventually returned to the sender with an error notification. Third, all channels are

FIFO. Specifically, a process receives a message from another process only after it has received all the messages sent to it earlier by that process. The first two assumptions are similar to those made by Dhamdhere *et al* [13].

We model execution of a process as an alternating sequence of *states* and *events*. A process changes its state by executing an event. Additionally, a send event causes a message to be sent and a receive event causes a message to be received. Sometimes, we refer to the state of a process as *local state*. To avoid confusion, we use the letters a, b, c, d, e and f to refer to events and the letters u, v, w, x, y and z to refer to local states.

Events on a process are totally ordered. However, events on different processes are only partially ordered by the Lamport's *happened-before* relation [17], which is defined as the smallest transitive relation satisfying the following properties:

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

For an event e, let $process(e)$ denote the process on which e is executed. Likewise, for a local state x, let $process(x)$ denote the process to which x belongs. We define $events(x)$ as the set consisting of all events that have to be executed to reach x. Intuitively, $events(x)$ captures the causal past of x.

A state of the system is given by the set of events that have been executed so far. We assume that existence of fictitious events $\perp$ that initialize the state of the system. Further, every collection (or set) of events we consider contains these initial events. Clearly, a collection of events E corresponds to a valid state of the system only if E is closed with respect to the happened-before relation. We refer to such a collection of events as *comprehensive cut*. Formally,

$$E \text{ is a comprehensive cut} \triangleq (\perp \subseteq E) \wedge \langle \forall e, f :: (f \in E) \wedge (e \rightarrow f) \Rightarrow e \in E \rangle$$

Sometimes, it is more convenient to model a system state using a collection of local states instead of using a collection of events, especially when taking a snapshot of the system. Intuitively, a comprehensive state is obtained by executing all events in a comprehensive cut. In this paper, we use the term "comprehensive cut" to refer to a collection of events and the term "comprehensive state" to refer to a collection of local states. To avoid confusion, we use the letters A, B, C, D, E and F to refer to a collection of events and the letters U, V, W, X, Y and X to refer to a collection of local states.

For a collection of local states X, let $processes(X)$ denote the set of processes whose local state is in X. Also, let $events(X)$ denote the set of events that have to be executed to reach local states in X.

Two local states x and y are said to be *consistent* if, in order to reach x on $process(x)$, we do not have to advance beyond y on $process(y)$, and vice versa.

Definition 1 (consistent collection of local states). *A collection of local states is said to be* consistent *if all local states in the collection are pair-wise consistent.*

Note that, for a collection of local states to form a comprehensive state, the local states should be pair-wise consistent. However, not every consistent collection of local states forms a comprehensive state. This happens when the collection is missing local states from certain processes. Specifically, a collection of local states X corresponds to a comprehensive state if the following two conditions hold: (1) X is consistent, and (2) X contains a local state from every process that has at least one event in $events(X)$.

For a consistent collection of local states X, let $CS(X)$ denote the system state obtained by executing all events in $events(X)$. Clearly, $X \subseteq CS(X)$ and, moreover, $CS(X)$ corresponds to a comprehensive state.

For a static distributed system, a system state is captured using the notion of consistent global state. A collection of local states forms a *consistent global state* if the collection is consistent and it contains one local state from every process in the system. For a dynamic distributed system, however, the set of processes may change with time. As a result, the term "every process in the system" is not well-defined. Therefore, we use a slightly different definition of system state, and, to avoid confusion, we use the term "comprehensive state" instead of the term "consistent global state" to refer to it.

2.2 Stable Properties

A property maps every comprehensive state of the system to a boolean value. Intuitively, a property is said to be stable if it stays true once it becomes true. For two comprehensive states X and Y, we say that Y lies in the future of X, denoted by $X \preccurlyeq Y$, if $events(X) \subseteq events(Y)$. Then, a property ϕ is stable if for every pair of comprehensive states X and Y,

$$(\phi \text{ holds for } X) \wedge (X \preccurlyeq Y) \;\; \Rightarrow \;\; \phi \text{ holds for } Y$$

We next describe an algorithm to detect a stable property in a dynamic distributed system.

3 Our Algorithm

3.1 The Main Idea

A common approach to detect a stable property in a *static* distributed system is to repeatedly collect a consistent set of local states, one from each process. Such a collection is also referred to as a *(consistent) snapshot* of the system. The property is then evaluated for the snapshot collected until it evaluates to true. The problem of collecting local states, one from each process, is relatively easier for a static distributed system than for a dynamic distributed system. This is because, in a static system, the set of processes is fixed and does not change with time. In a dynamic system, however, the set of processes may change with time. Therefore it may not always be clear local states of which processes have to be included in the collection.

In our approach, we impose a logical spanning tree on processes that are currently part of the system. The spanning tree is used to collect local states

of processes currently attached to the tree. Observe that, to be able to evaluate the property, the collection has to at least include local states of all processes that are currently part of the application. Therefore we make the following two assumptions. First, a process attaches itself to the spanning tree *before* joining the application. Second, a process leaves the application *before* detaching itself from the spanning tree.

A process joins the spanning tree by executing a *control join protocol* and leaves the spanning tree by executing a *control depart protocol*. Likewise, a process joins the application by executing an *application join protocol* and leaves the application by executing an *application depart protocol*.

We associate a status with every process, which can either be OUT, JOINING, IN, TRYING, DEPARTING. Intuitively, status captures the state of a process *with respect to the spanning tree*. A process that is not a part of the system (that is, before it starts executing the control join protocol or after it has finished executing the control depart protocol) has status OUT. When a process starts executing its control join protocol, its status changes to JOINING. The status changes to IN once the join protocol finishes and the process has become part of the spanning tree. When a process wants to leave the spanning tree, it begins executing the control depart protocol, which consists of two parts. In the first part, the process tries to obtain permission to leave from all its neighboring processes. In the second part, it actually leaves the spanning tree. But, before leaving the system, it ensures that the set of processes currently in the system remain connected. During the former part of the depart protocol, its status is TRYING and, during the latter part, its status is DEPARTING.

Typically, for evaluating a property, state of a process can be considered to consist of two components. The first component captures values of all program variables on a process; we refer to it as *core* state. The second component is used to determine state of a channel (*e.g.*, the number of messages a process has sent to another process); we refer to it as *non-core* state. We assume that, once a process has detached itself from the application its core state is no longer needed to evaluate the property. However, its non-core state may still be required to determine the *state of an outgoing channel* it has with another process that is still part of the application. For example, consider a process p that leaves the application soon after sending an application message m to process q. In this case, m may still be in transit towards q after p has left the application. If q does not know about the departure of p when it receives m and it is still part of the application, then it has to receive and process m. This may cause q's core state to change, which, in turn, may affect the value of the property. In this example, even though p has left the application, its non-core state is required to determine the state of the channel from p to q, which is non-empty.

We say that an application message is *irrelevant* if either it is never delivered to its destination process (and is therefore returned to the sender with error notification) or when it is delivered, its destination process is no longer part of the application; otherwise the message is *relevant*. In order to prevent the aforementioned situation from arising, we make the following assumption about an application depart protocol:

Assumption 1. *Once a process has left the application, none of its outgoing channels, if non-empty, contains a relevant application message.*

The above assumption can be satisfied by using acknowledgments for application messages. Specifically, a process leaves the application only after ensuring that, for every application message it sent, it has either received an acknowledgment for it or the message has been returned to it with error notification. Here, we assume that a process that is no longer a part of the application, on receiving an application message, still sends an acknowledgment for it. It can be verified that this scheme implements Assumption 1.

Assumption 1 is useful because it enables a process to evaluate a property using local states of only those processes that are currently part of the spanning tree. Specifically, to evaluate the property, a process does not need information about states of processes that left the system before the snapshot algorithm started.

Now, to understand local states of which processes need to be recorded in a snapshot, we define the notion of *completeness*. We call a process *active* if its status is IN and *semi-active* if its status is either IN or TRYING. Further, for a collection of local states X, let $active(X)$ denote the set of local states of all those processes whose status is IN in X. We can define $semi\text{-}active(X)$ similarly.

Definition 2 (complete collection of local states). *A consistent collection of local states Y is said to be* complete *with respect to a comprehensive state X with $Y \subseteq X$ if Y includes local states of all those processes whose status is IN in X. Formally,*

$$Y \text{ is complete with respect to } X \quad \triangleq \quad active(X) \subseteq Y$$

From Assumption 1, to be able to evaluate a property for a collection of local states, it is sufficient for the collection to be complete; it need not be comprehensive. This is also important because our definition of comprehensive state includes local states of even those processes that are no longer part of the system. As a result, if a snapshot algorithm were required to return a comprehensive state, it will make the algorithm too expensive. As we see later, our snapshot algorithm returns a collection that contains local states of *all semi-active processes* of some comprehensive state (and not just all active processes).

3.2 Spanning Tree Maintenance Algorithm

Processes may join and leave the system while an instance of the snapshot algorithm is in progress. Therefore spanning tree maintenance protocols, namely control join and depart protocols, have to designed carefully so that they do not "interfere" with an ongoing instance of the snapshot algorithm. To that end, we maintain a set of invariants that we use later to establish the correctness of the snapshot algorithm.

Each process maintains information about its parent and its children in the tree. Initially, before a process joins the spanning tree, it does not have any parent or children, that is, its parent variable is set to nil and its children-set is empty. Let x be a local state of process p. We use $parent(x)$ to denote the parent of p in x and $children(x)$ to denote the set of children of p in x. Also, let $status(x)$ denote the status of p in x. Further, p is said to be *root* in x if

$parent(x) = p$. For a collection of local states X and a process $p \in processes(X)$, we use $X.p$ to denote the local state of p in X.

Now, we describe our invariants. Consider a comprehensive state X and let p and q be two processes in X. The first invariant says that if the status of a process is either IN or TRYING, then its parent variable should have a non-nil value. Formally,

$$status(X.p) \in \{\mathsf{IN}, \mathsf{TRYING}\} \quad \Rightarrow \quad parent(X.p) \neq nil \tag{1}$$

The second invariant says that if a process considers another process to be its parent then the latter should consider the former as its child. Moreover, the parent variable of the latter should have a non-nil value. Intuitively, it means that child "relationship" is maintained for a longer duration than parent "relationship". Further, a process cannot set its parent variable to nil as long as there is at least one process in the system, different from itself, that considers it to be its parent. Formally,

$$\begin{aligned}(parent(X.p) = q) \wedge (p \neq q) \;\Rightarrow \\ (p \in children(X.q)) \wedge (parent(X.q) \neq nil)\end{aligned} \tag{2}$$

The third invariant specifically deals with the departure of a root process. To distinguish between older and newer root processes, we associate a *rank* with every root process. The rank is incremented whenever a new root is selected. This invariant says that if two processes consider themselves to be root of the spanning tree, then there cannot be a process that considers the "older" root to be its parent. Moreover, the status of the "older" root has to be DEPARTING. Formally,

$$\begin{aligned}root(X.p) \wedge root(X.q) \wedge (rank(X.p) < rank(X.q)) \;\Rightarrow \\ \langle \nexists r : r \in processes(X) \setminus \{p\} : parent(X.r) = p \rangle \wedge \\ (status(X.p) = \mathsf{DEPARTING})\end{aligned} \tag{3}$$

We now describe our control join and depart protocols that maintain the invariants (1)–(3).

Joining the Spanning Tree: A process attaches itself to the spanning tree by executing the control join protocol. Our control join protocol is quite simple. A process wishing to join the spanning tree first obtains a list of processes that are currently part of the spanning tree. This, for example, can be achieved using a name server. It then contacts the processes in the list, one by one, until it finds a process that is willing to accept it as its child. We assume that the process is eventually able to find such a process, and, therefore, the control join protocol eventually terminates successfully.

Leaving the Spanning Tree: A process detaches itself from the spanning tree by executing the control depart protocol. The protocol consists of two phases. The first phase is referred to as *trying* phase and the status of process in this phase is TRYING. In the trying phase, a departing process tries to obtain permission to leave from all its tree neighbors (parent and children). To prevent

neighboring processes from departing at the same time, all departure requests are assigned timestamps using logical clock. A process, on receiving departure request from its neighboring process, grants the permission only if it is not departing or its depart request has larger timestamp than that of its neighbor. This approach is similar to Ricart and Agrawala's algorithm [18] modified for drinking philosopher's problem [19]. Note that the neighborhood of a departing process may change during this phase if one of more of its neighbors are also trying to depart. Whenever the neighborhood of a departing process changes, it sends its departure request to all its new neighbors, if any. A process wishing to depart has to wait until it has received permission to depart from its *current* neighbors.

We show in [16] that the first phase of the control depart protocol eventually terminates. Once that happens, the process enters the second phase. The second phase is referred to as *departing* phase and the status of process in this phase is DEPARTING. The protocol of the departing phase depends on whether the departing process is a root process. If the departing process is not a root process, then, to maintain the spanning tree, it attaches all its children to its parent. On the other hand, if it is a root process, then it selects one its children to become the new root. It then attaches all its other children to the new root. The main challenge is to change the spanning tree without violating any of the invariants.

Case 1 (when the departing process is not the root): In this case, the departing phase consists of the following steps:

- **Step 1:** The departing process asks its parent to inherit all its children and waits for acknowledgment.
- **Step 2:** The departing process asks all its children to change their parent to its parent and waits for acknowledgment from all of them. At this point, no process in the system considers the departing process to be its parent.
- **Step 3:** The departing process terminates all its neighbor relationships. At this point, the parent of the departing process still considers the process to be its child.
- **Step 4:** The departing process asks its parent to remove it from its set of children and waits for acknowledgment.

Case 2 (when the departing process is the root): In this case, the departing phase consists of the following steps:

- **Step 1:** The departing process selects one of its children to become the new root. It then asks the selected child to inherit all its other children and waits for acknowledgment.
- **Step 2:** The departing process asks all its other children to change their parent to the new root and waits for acknowledgment from all of them. At this point, only the child selected to become the new root considers the departing process to be its parent.
- **Step 3:** The departing process terminates child relationships with all its other children. The child relationship with the child selected to become the new root cannot be terminated as yet.
- **Step 4:** The departing process asks the selected child to become the new root of the spanning tree and waits for acknowledgment. At this point, no process in the system considers the departing process to be its parent.

– **Step 5:** The departing process terminates all its neighbor relationships.

To ensure liveness of the snapshot algorithm, we require the departing process to "transfer" the latest set of local states it has collected so far (which may be empty) to another process, after it has detached itself from the spanning tree but before leaving the system permanently. The process to which the collection has to be "transfered" is the parent of the departing process in the first case and the new root of the spanning tree in the second case. In both cases, the process to which the collection is "transfered" has to wait until it has received the collection from all processes it is supposed to before it can itself enter the departing phase.

3.3 The Snapshot Algorithm

As discussed earlier, it is sufficient to collect a *consistent* set of local states that is *complete* with respect to some comprehensive state. We next discuss how consistency and completeness can be achieved. For convenience, when a process records it local state, we say that it has taken its snapshot.

Achieving Consistency: To achieve consistency, we use Lai and Yang's approach for taking a consistent snapshot of a static distributed system [2]. Each process maintains the *instance number* of the latest snapshot algorithm in which it has participated. This instance number is piggybacked on every message it sends—application as well as control. If a process receives a message with an instance number greater than its own, it first records its local state before delivering the message. It can be verified that:

Theorem 3 (consistency). *Two local states belonging to the same instance of the snapshot algorithm are consistent with each other.*

Achieving Completeness: As explained earlier in Sect. 3.1, to be able to evaluate a property for a collection of local states, it is sufficient for the collection to be complete with respect to some comprehensive state. The main problem is: *"How does the current root of the spanning tree know that its collection has become complete?"* To solve this problem, our approach is to define a *test property* that can be evaluated *locally* for a collection of local states such that once the test property evaluates to true then the collection has become complete. To that end, we define the notion of *f-closed* collection of local states.

Definition 4 (f-closed collection of local states). *Let f be a function that maps every local state to a set of processes. A consistent collection of local states X is said to be* f-closed *if, for every local state x in X, X contains a local state from every process in $f(x)$. Formally,*

$$X \text{ is } f\text{-closed} \quad \triangleq \quad \langle \forall x \in X :: f(x) \subseteq processes(X) \rangle$$

Intuitively, f denotes a neighborhood function. For example, f may map a local state x to $children(x)$. We consider two special cases for function f. For a

local state x, let $\rho(x)$ be defined as the set containing the parent of $process(x)$ in local state x, if it exists. Further, let $\kappa(x)$ be defined as the set of children of $process(x)$ in local state x, that is, $\kappa(x) = children(x)$. We show that, under certain condition, if the collection is $(\rho \cup \kappa)$-closed, then it is also complete. To capture the condition under which this implication holds, we define the notion of f-path as follows:

Definition 5 (f-path). *Let f be a function that maps every local state to a set of processes. Consider a comprehensive state X and two distinct processes p and q in $processes(X)$. We say that there is an f-*path *from p to q in X, denoted by $f\text{-}path(p, q, X)$, if there exists a sequence of processes $s_i \in processes(X)$ for $i = 1, 2, \ldots, m$ such that:*

1. *$s_1 = p$ and $s_m = q$*
2. *for each i, $1 \leq i < m$, $s_i \neq s_{i+1}$ and $s_{i+1} \in f(X.s_i)$*

Using the notion of f-path, we define the notion of an f-connected state as follows:

Definition 6 (f-connected state). *Let f be a function that maps every local state to a set of processes. A comprehensive state X is said to be f-*connected *if there is a f-path between every pair of distinct processes in $semi\text{-}active(X)$. Formally, X is f-connected if*

$$\langle \forall p, q \in processes(X) : p \neq q : \{p, q\} \subseteq semi\text{-}active(X) \;\Rightarrow\; f\text{-}path(p, q, X) \rangle$$

Using the invariants (1)–(3), we show that every comprehensive state is actually $(\rho \cup \kappa)$-connected. We first prove an important property about the spanning tree maintained by our algorithm.

Theorem 7. *The directed graph induced by parent variables of a comprehensive state is acyclic (except for self-loops).*

The following theorem can now be proved:

Theorem 8. *Every comprehensive state is $(\rho \cup \kappa)$-connected.*

The main idea behind the proof is to show that each semi-active process has a *ρ-path* to the current root of the spanning tree. This, in turn, implies that there is a *κ-path* from the current root to each semi-active process in the system. We now provide a sufficient condition for a collection of local states to be complete.

Theorem 9 (f-closed and f-connected $\Rightarrow$ complete). *Let f be a function that maps every local state to a set of processes. Consider a consistent collection of local states X. If (1) X is f-closed, (2) $semi\text{-}active(X) \neq \emptyset$, and (3) $CS(X)$ is f-connected, then X is complete with respect to $CS(X)$.*

Therefore it suffices to ensure that the set of local states collected by the snapshot algorithm is $(\rho \cup \kappa)$-closed. We now describe our snapshot algorithm. After recording its local state, a process waits to receive local states of its children in the tree until its collection becomes κ-closed. As soon as that happens, it sends

the collection to its (current) parent in the spanning tree unless it is a root. In case it is a root, it uses the collection to determine whether the property of interest (*e.g.*, termination) has become true. A root process initiates the snapshot algorithm by recording its local state provided its status is either IN or TRYING. This ensures that the collection contains a local state of at least one semi-active process. (Note that the snapshot algorithm described above does not satisfy liveness. We describe additions to the basic snapshot algorithm to ensure its liveness later.) The next theorem establishes that the collection of local states returned by an instance of the snapshot algorithm is not only κ-closed but also $(\rho \cup \kappa)$-closed.

Theorem 10. *The collection of local states returned by the snapshot algorithm is consistent and $(\rho \cup \kappa)$-closed.*

It follows from Theorem 3, Theorem 9, Theorem 8 and Theorem 10 that:

Corollary 11 (safety). *The collection of local states returned by the snapshot algorithm is (1) consistent and (2) complete with respect to some comprehensive state.*

The liveness of the snapshot algorithm is only guaranteed if the system becomes permanently quiescent eventually (that is, the set of processes does not change). Other algorithms for property detection make similar assumptions to achieve liveness [13, 15]. Without this assumption, the spanning tree may continue to grow forcing the snapshot algorithm to collect local states of an ever increasing number of processes. To ensure liveness under this assumption, we make the following enhancements to the basic snapshot algorithm. First, whenever a process records its local state, it sends a marker message containing the current instance number to all its neighbors. In addition, it sends a marker message to any new neighbor whenever its neighborhood set changes. Second, whenever its parent changes, it sends its collection to the new parent if the collection has become κ-closed. Third, just before leaving the system, a process transfers its collection to one of its neighbors as explained earlier. Once the system becomes permanently quiescent, the first modification ensures that all processes in the tree eventually record their local states and the second modification ensures that the collection at the root eventually becomes κ-closed. It can be proved that:

Theorem 12 (liveness). *Assuming that the system eventually becomes permanently quiescent (that is, the set of processes does not change), every instance of the snapshot algorithm terminates eventually.*

4 Conclusion and Future Work

In this paper, we present an efficient algorithm to determine whether a stable property has become true in a dynamic distributed system in which processes can join and leave the system at any time. Our approach involves periodically collecting local states of processes that are currently part of the system using a (dynamically changing) spanning tree.

There are several interesting problems that still need to be addressed. The depart protocol described in the paper has relatively high worst-case depart

latency. Specifically, a process may stay in the trying phase for a long period of time (because of other processes joining and leaving the system) before it is able to enter the departing phase. An interesting problem is to design a depart protocol that has low worst-case depart latency. Also, in our current approach, control neighbors of a process may be completely different from its application neighbors, which may be undesirable in certain cases. Finally, in this paper, we assume that processes are reliable and they never fail. It would be interesting to investigate this problem in the presence of failures.

References

1. Chandy, K.M., Lamport, L.: Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems **3** (1985) 63–75
2. Lai, T.H., Yang, T.H.: On Distributed Snapshots. Information Processing Letters (IPL) **25** (1987) 153–158
3. Alagar, S., Venkatesan, S.: An Optimal Algorithm for Recording Snapshots using Casual Message Delivery. Information Processing Letters (IPL) **50** (1994) 311–316
4. Dijkstra, E.W., Scholten, C.S.: Termination Detection for Diffusing Computations. Information Processing Letters (IPL) **11** (1980) 1–4
5. Francez, N.: Distributed Termination. ACM Transactions on Programming Languages and Systems (TOPLAS) **2** (1980) 42–55
6. Ho, G.S., Ramamoorthy, C.V.: Protocols for Deadlock Detection in Distributed Database Systems. IEEE Transactions on Software Engineering **8** (1982) 554–557
7. Chandy, K.M., Misra, J., Haas, L.M.: Distributed Deadlock Detection. ACM Transactions on Computer Systems **1** (1983) 144–156
8. Marzullo, K., Sabel, L.: Efficient Detection of a Class of Stable Properties. Distributed Computing (DC) **8** (1994) 81–91
9. Schiper, A., Sandoz, A.: Strong Stable Properties in Distributed Systems. Distributed Computing (DC) **8** (1994) 93–103
10. Atreya, R., Mittal, N., Garg, V.K.: Detecting Locally Stable Predicates without Modifying Application Messages. In: Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS). (2003) 20–33
11. distributed.net: http://www.distributed.net/projects.php (2005)
12. Lai, T.H.: Termination Detection for Dynamic Distributed Systems with Non-First-In-First-Out Communication. Journal of Parallel and Distributed Computing (JPDC) **3** (1986) 577–599
13. Dhamdhere, D.M., Iyer, S.R., Reddy, E.K.K.: Distributed Termination Detection for Dynamic Systems. Parallel Computing **22** (1997) 2025–2045
14. Wang, X., Mayo, J.: A General Model for Detecting Termination in Dynamic Systems. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS), Santa Fe, New Mexico (2004)
15. Darling, D., Mayo, J.: Stable Predicate Detection in Dynamic Systems. Submitted to the Journal of Parallel and Distributed Computing (JPDC) (2003)
16. Peri, S., Mittal, N.: Monitoring Stable Properties in Dynamic Peer-to-Peer Distributed Systems. Technical Report UTDCS-27-05, Department of Computer Science, The University of Texas at Dallas, Richardson, TX, 75083, USA (2005)
17. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM (CACM) **21** (1978) 558–565
18. Ricart, G., Agrawala, A.K.: An Optimal Algorithm for Mutual Exclusion in Computer Networks. Communications of the ACM (CACM) **24** (1981) 9–17
19. Chandy, K.M., Misra, J.: The Drinking Philosophers Problem. ACM Transactions on Programming Languages and Systems (TOPLAS) **6** (1984) 632–646