# Monitoring Stable Properties in Dynamic Asynchronous Distributed Systems*

Sathya Peri 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 inference about the system state. In contrast to existing algorithms for stable property detection in a dynamic environment, our algorithm is general in the sense that it can be used to detect *any* stable property and not just a specific stable property such as termination.

## 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

*A preliminary version of the paper first appeared in the 2005 IARCS Annual Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS)

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 studied extensively and numerous solutions have been proposed for solving the general problem (*e.g.*, [CL85, LY87, AV94]) as well as its special cases (*e.g.*, [DS80, Fra80, HR82, CMH83, MS94, SS94, AMG03]). 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 [dis] 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 [Lai86, DIR97, WM04] or can be expressed as conjunction of local predicates on process states [DMW05]. (Darling *et al.* [DMW05] also describe an extension to their basic algorithm to handle predicates on channel states. However, they assume that channel states stop changing eventually.) Second, most of the algorithms assume the existence of permanent processes that never leave the system [DIR97, DMW05, WM04]. Further, Dhamdhere *et al.* [DIR97] assumes that all initial processes are permanent processes, which is stronger than assuming that there is at least one permanent process in the system. Third, some of the algorithms assume that processes can join but cannot leave the system until the property has become true and the detection algorithm has terminated [Lai86, WM04]. Fourth, the algorithm by Darling *et al.* [DMW05] assumes that processes are equipped with local clocks that are weakly synchronized, which can be achieved using GPS (global positioning system) receivers. Table 1 compares various stable property detection algorithms that have been proposed for dynamic distributed systems.

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 span-

| Algorithm | Assumptions | |
|---|---|---|
| | Dynamism in the System | Stable Property to be Detected |
| Lai [Lai86] | join only | termination |
| Dhamdhere *et al.* [DIR97] | all initial processes are permanent | termination |
| Wang and Mayo [WM04] | join only | termination |
| Darling *et al.*$^*$ [DMW05] | there is at least one permanent process | conjunction of local predicates on process states$^\dagger$ |
| Our algorithm [this paper] | - | - |

$^*$: also assumes that local clocks of various processes are weakly synchronized

$^\dagger$: can be relaxed to include predicates on channel states as well

Table 1: Comparison of various stable property detection algorithms.

ning 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.

This paper is organized as follows. We discuss the system model and notation used in this paper in Section 2. The main idea behind our approach is described in Section 3. The spanning tree maintenance algorithm is described in Section 4 and the snapshot algorithm is discussed in Section 5. We analyze the complexity of our approach in Section 6. Finally, we present our observations and outline directions for future research in Section 7.

# 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 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* with 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 (first-in-first-out). 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.* [DIR97].

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 [Lam78], 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 $f$.

Given 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. Also, let $last(x)$ denote the *last* event executed by $process(x)$ before reaching $x$. We define $events(x)$ as the set consisting of all events that have to be executed to reach $x$. In other words, $events(x)$ includes the event $last(x)$

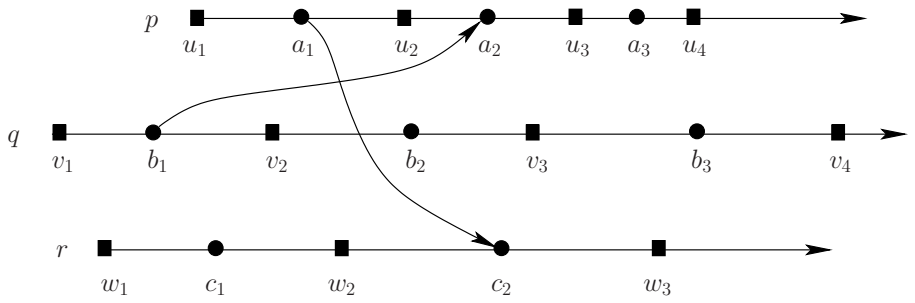


Figure 1: An execution of a distributed system.

and all events that happened-before $last(x)$. Formally,

$$events(x) \quad \triangleq \quad \{e \mid e = last(x) \text{ or } e \rightarrow last(x)\}$$

Intuitively, $events(x)$ captures the causal past of $x$.

**Example 1** Consider an execution of a distributed system shown in Figure 1. In the figure, local states are represented using solid squares and events are represented using solid circles. For the execution in the figure, $process(a_1) = p$ and $process(b_1) = q$. Also, $process(u_1) = p$ and $process(v_1) = q$. Further, $last(u_3) = a_2$ and $events(u_3) = \{a_1, a_2, b_1\}$. □

## 2.2 System States

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,

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

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. In this paper, we use the state-based model to describe the snapshot algorithm, as is typical of snapshot algorithms. However, we use a combination of

| Symbols | Meaning |
|---|---|
| $p$, $q$, $r$, $s$ and $t$ | processes |
| $a$, $b$, $c$, $d$, $e$ and $f$ | events |
| $u$, $v$, $w$, $x$, $y$ and $z$ | local states |
| $A$, $B$, $C$, $D$, $E$ and $F$ | collections of events |
| $U$, $V$, $W$, $X$, $Y$ and $Z$ | collections of local states |

Table 2: Symbols used in the paper.

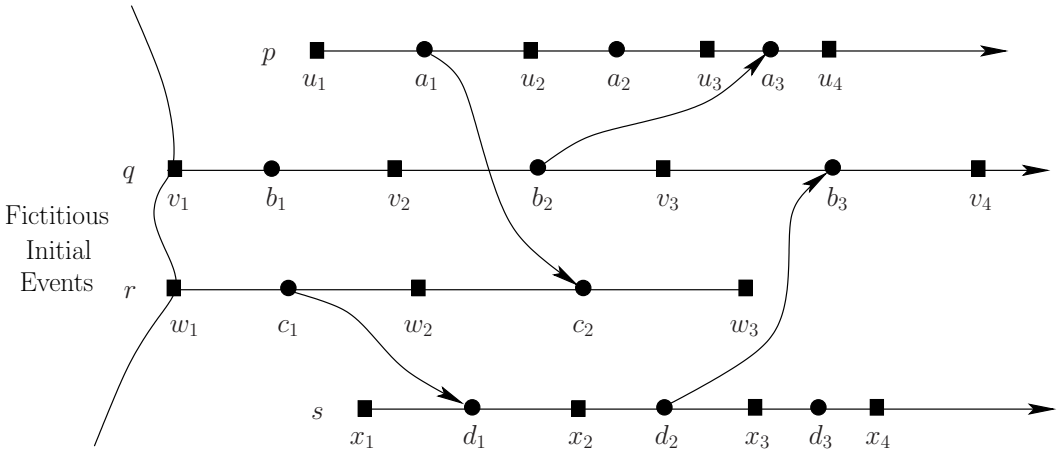


Figure 2: An illustration of the execution of a dynamic distributed system.

state- and event-based models to prove its correctness. 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.

**Example 2** Consider the execution of a distributed system depicted in Figure 2. We do not show the fictitious initial events in the figure. Processes $q$ and $r$ are present in the system initially, whereas processes $p$ and $s$ join later. Process $q$ leaves the system at state $w_3$.

Consider the collection of events $A = \perp \cup \{a_1, a_2, b_1, c_1\}$, $B = \perp \cup \{b_1, c_1, d_1, d_2\}$ and $C = \perp \cup \{a_1, a_2, a_3, b_1\}$. The collections $A$ and $B$ correspond to comprehensive cuts, whereas the collection $C$ does not. Now, consider the the collections of local states $U = \{u_1, v_1, w_2, x_2\}$, $V = \{v_1, w_2, x_3\}$ and $W = \{v_1, w_3, x_1\}$. The collections $U$ and $V$ form comprehensive states, whereas the collection $W$ does not. $W$ is not a comprehensive state because, to reach the local state $w_3$, the event $a_1$ on process $p$ has to be executed. Therefore, to be comprehensive, the collection should include some local state of $p$ as well. □

Given a collection of events $E$, let *process-set*$(E)$ denote the set of processes with at least one event in $E$, that is,

$$\textit{process-set}(E) \quad \triangleq \quad \{\textit{process}(e) \mid e \in E\}$$

Also, let *states*$(E)$ denote the collection of local states obtained after executing all events in $E$. Observe that *states*$(E)$ contains one local state from every process in *process-set*$(E)$.

Given a collection of local states $X$, let *process-set*$(X)$ denote the set of processes whose local state is in $X$, that is,

$$\textit{process-set}(X) \quad \triangleq \quad \{\textit{process}(x) \mid x \in X\}$$

Also, let *events*$(X)$ denote the set of events that have to be executed to reach local states in $X$, that is,

$$\textit{events}(X) \quad \triangleq \quad \bigcup_{x \in X} \textit{events}(x)$$

**Example 3** Consider the collections of events and local states defined in Example 2 for the execution shown in Figure 2. Clearly, *process-set*$(A) = \{p, q, r\}$ and *states*$(A) = \{u_3, b_2, w_2\}$. Also, *process-set*$(U) = \{p, q, r, s\}$ and *events*$(U) = \perp \cup \{c_1, d_1\}$. □

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. Formally, $x$ and $y$ are consistent if

$$\langle \nexists e : e \in \textit{events}(x) \setminus \textit{events}(y) : \textit{process}(e) = \textit{process}(y) \rangle \bigwedge$$
$$\langle \nexists e : e \in \textit{events}(y) \setminus \textit{events}(x) : \textit{process}(e) = \textit{process}(x) \rangle$$

**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*$(X)$.

Given 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.

**Remark 1** 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. □

### 2.3 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. Given 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 $\phi$ is stable if for every pair of comprehensive states $X$ and $Y$,

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

Our focus in the paper is on detecting whether a stable property has become true. For example, whether a distributed computation has terminated.

## 3 Our Approach: 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.

### 3.1 Spanning Tree of Processes

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

We associate a status with every process, which can either be OUT, ATTACHING, IN, TRYING, DETACHING. 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 operation or after it has finished executing the control depart operation) has status OUT. When a process starts executing its control join operation, its status changes to ATTACHING. The status changes to IN once the join operation 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 operation, 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 operation, its status is TRYING and, during the latter part, its status is DETACHING. The state transition diagram of a process is shown in Figure 3.

We assume that the application is *well-behaved* in the sense that a process that is part of the application can send an application message to another process only if the latter is currently or has been a part of the application. In other words, *an application message cannot be sent to a process that is still not part of the application.*

Note that, in practice, application depart operation will usually ensure that, once a process has left the system, no application message is sent to it thereafter. We make a weaker assumption to make our approach more general.

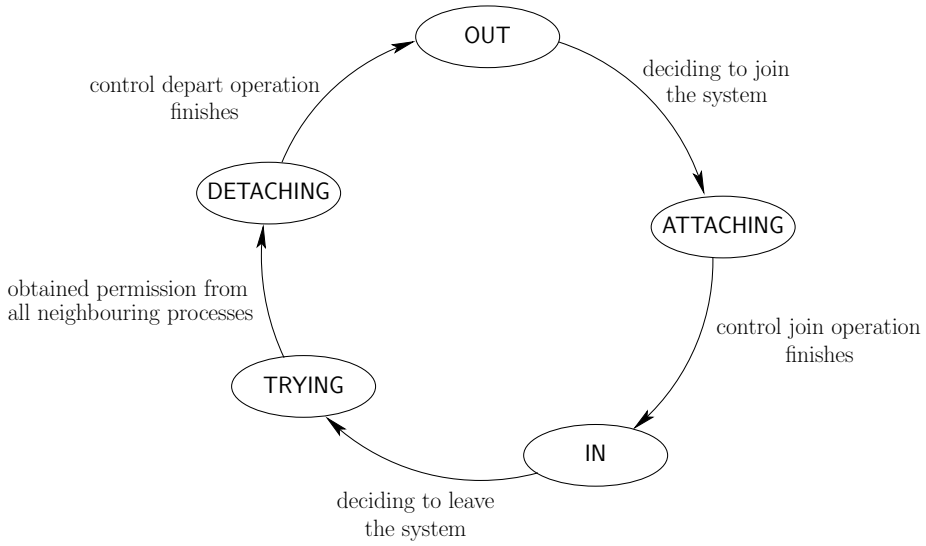


Figure 3: State transition diagram of a process.

### 3.2 Evaluating the Property

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 is still part of the application at that time, then it has to deliver and process $m$, that is, it cannot just ignore $m$. This may cause $q$'s core state to change, which, in turn, may affect the value of the property being monitored. 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 operation:

**Assumption 1** *Once a process has left the application, none of its outgoing channels, if non-empty, contains a relevant application message. Formally, for every comprehensive state $X$, process $p$ and application message $m$,*

$$(p \textit{ has left the application in } X) \wedge (p \textit{ sent } m) \wedge (m \textit{ is in transit with respect to } X)$$

$$\Rightarrow$$

$$m \textit{ is irrelevant}$$

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. 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.

**Lemma 1** *The scheme described above implements Assumption 1.*

Note that there are alternative ways of implementing Assumption 1 and we have only discussed one of them. For instance, as part of the application depart operation, a departing process can inform all processes that are its neighbors with respect to the application that it is leaving the system and wait for an acknowledgment from them. Once a process has received such a message from a departing process, it can simply ignore any application message it receives from the departing process later.

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*. Let $x$ be a local state of a process $p$. We call $p$ *active* in $x$ if its status is IN in $x$ and *semi-active* in $x$ if its status is either IN or TRYING in $x$. Formally,

$$active(x) \quad \triangleq \quad status(x) = \mathsf{IN} \tag{1}$$

$$semiactive(x) \quad \triangleq \quad status(x) \in \{\mathsf{IN}, \mathsf{TRYING}\} \tag{2}$$

Given a collection of local states $X$, let *active-set*$(X)$ denote the set of all those processes whose status is IN in $X$. We can similarly define *semiactive-set*$(X)$.

**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,*

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

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).

To prove the correctness of our algorithms, we define several properties which may be either "local" involving a single process or "global" involving the entire (or part of the) system. Whenever possible, we define a property to be local (that is, as a function of local state) and define a property to be global only if necessary.

# 4 The Spanning Tree Maintenance Algorithm

We would like processes to be able to join and leave the system while an instance of the snapshot algorithm is in progress. Therefore spanning tree maintenance algorithm, which consists of control join and depart operations, has to designed carefully so that it does not "interfere" with an ongoing instance of the snapshot algorithm. Our objective is to allow processes to join and/or depart the system while a snapshot algorithm is in progress. 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*$(x) = p$. Given a

collection of local states $X$ and a process $p \in \mathit{process\text{-}set}(X)$, 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,

$$\mathit{status}(X.p) \in \{\mathsf{IN}, \mathsf{TRYING}\} \quad \Rightarrow \quad \mathit{parent}(X.p) \neq \mathit{nil} \tag{I1}$$

The second invariant says that if a process is not a part of the spanning tree, then its parent should be set to nil and it should not have any children. Formally,

$$\mathit{status}(X.p) \in \{\mathsf{OUT}, \mathsf{ATTACHING}\} \quad \Rightarrow \quad (\mathit{parent}(X.p) = \mathit{nil}) \wedge (\mathit{children}(X.p) = \emptyset) \tag{I2}$$

The third 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,

$$(\mathit{parent}(X.p) = q) \ \wedge \ (p \neq q) \quad \Rightarrow \quad (p \in \mathit{children}(X.q)) \wedge (\mathit{parent}(X.q) \neq \mathit{nil}) \tag{I3}$$

The fourth invariant specifically deals with the departure of a root process. It is possible that when a root decides to depart from the system, one or more of the older roots may not have completely departed from the system. As a result, there may be multiple processes in the system all of which consider themselves to be root. 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 DETACHING. Formally,

$$\begin{aligned} &\mathit{root}(X.p) \wedge \ \mathit{root}(X.q) \ \wedge \ (\mathit{rank}(X.p) < \mathit{rank}(X.q)) \quad \Rightarrow \\ &\qquad \langle \nexists r : r \in \mathit{process\text{-}set}(X) \setminus \{p\} : \mathit{parent}(X.r) = p \rangle \ \wedge \ (\mathit{status}(X.p) = \mathsf{DETACHING}) \end{aligned} \tag{I4}$$

We now describe our control join and depart operations that maintain the invariants (I1)–(I4).

```
Actions pertinent to the control join operation for process p:

Variables:
    status: state of the process with respect to the spanning tree;
    parent: parent in the spanning tree;
    children: set of children in the spanning tree;

(J1) On deciding to join the spanning tree:
    status := ATTACHING;
    identify a process q in the system (e.g., using a name server);
    send JOIN message to process q;

(J2) On receiving JOIN message from process q:
    if (status ≠ IN) then
        // reject q's request to join the tree
        send REJECT message to q;
    else
        // accept q's request to join the tree
        send ACCEPT message to q;
        add q to children;
    endif;

(J3) On receiving either REJECT message from process q or
                        error notification for the JOIN message sent to process q:
    // request to join the tree failed
    identify another process r in the system (e.g., using a name server);
    send JOIN message to process r;

(J4) On receiving ACCEPT message from process q:
    // attach to the spanning tree
    parent := q;
    status := IN;
```

Figure 4: Actions pertinent to the control join operation.

### 4.1 Joining the Spanning Tree

A process attaches itself to the spanning tree by executing the control join operation. We assume the existence of *name server*, and any process that wishes to join the system can contact to it obtain a list of processes currently in the system. This assumption is made by other algorithms for dynamic distributed systems as well (*e.g.*, [LMP04, DMW05]). Our control join operation is quite simple. A process wishing to join the spanning tree first contacts the name sever and 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 operation eventually terminates successfully.

Baldoni *et al.* [BHP04] prove that it is impossible to ensure the liveness of join operation unless there is permanent process in the system. In practice, however, if processes stay in the application for "sufficiently long" time, then a process, wishing to join the system, should be eventually able to locate a process that is willing to accept it as its child. In Darling *et al.*'s approach [DMW05], a process can join the system only through a process whose local predicate is currently false. On the other hand, in our approach, a process can join the system through any process that is still part of the application irrespective of what its current state is with respect to the application. With our approach, it is sufficient for an incoming process to locate a process that is still part of the application. With Darling *et al.*'s approach, however, an incoming process has to locate a process that is not only part of the application but whose local predicate currently evaluates to false as well. Therefore an incoming process may end up contacting the same process more than once.

### 4.2 Leaving the Spanning Tree

A process detaches itself from the spanning tree by executing the control depart operation. We divide the operation into 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 (or depart) 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. (Any ties are broken using process identifiers.) This approach is similar to Ricart and Agrawala's algorithm [RA81] modified for drinking philosopher's problem [CM84]. 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 later show that the first phase of the control depart operation eventually terminates. Once that happens, the process enters the second phase. The second phase is referred to as *detaching* phase and the status of process in this phase is DETACHING. The actions of the detaching 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

Actions pertinent to the trying of the control depart operation for process $p$:

Useful expressions:
$neighbors \triangleq children \cup \{ parent \mid parent \neq nil$ and $parent \neq p\}$

(T1) On deciding to leave the spanning tree:
$status :=$ TRYING;
*// assign a logical timestamp to the departure request*
$departTS :=$ current value of the logical clock;
*// ask for permission to depart from all neighboring processes*
send REQUEST$\langle departTS \rangle$ message to all processes in $neighbors$;

(T2) On receiving REQUEST$\langle nhbrDepartTS \rangle$ message from process $q$:
**if** $(status =$ IN$)$ **or**
$((status =$ TRYING$)$ and $(departTS > nhbrDepartTS))$ **then**
*// $q$'s departure request has higher priority than my own departure request, if any*
send GRANT message to process $p$;
**endif**;

(T3) On receiving GRANT message from process $q$
**if** (GRANT message has been received from all processes in $neighbors$) **then**
*// can depart safely*
$status :=$ DETACHING;
**endif**;

(T4) On addition of new processes to $neighbors$:
**if** $(status =$ TRYING$)$ **then**
*// ask for permission to depart from new neighbors*
send REQUEST$\langle departTS \rangle$ message to all new processes in $neighbors$;
**endif**;

Figure 5: Actions pertinent to the trying phase of the control depart operation.

all its other children to the new root. The main challenge is to change the spanning tree without violating any of the invariants.

### 4.2.1 Case 1 (when the departing process is not the root)

In this case, the detaching 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.

A formal description of the detaching phase for this case is given in Figure 6. Figure 7 depicts a step-by-step illustration of the detaching phase.

#### 4.2.2 Case 2 (when the departing process is the root)

In this case, the detaching 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.

A formal description of the detaching phase for this case is given in Figure 8. Figure 9 depicts a step-by-step illustration of the detaching phase.

#### 4.2.3 Proving Liveness of Depart Operation

Observe that our rule for transition from the trying phase to the detaching phase ensures that, once a process enters the detaching phase, none of its neighbors can be in the detaching phase. As a result, detaching phases of two processes do not interfere with each other and a departing

Actions pertinent to the detaching phase of the control depart operation for process $p$:
(Case 1: when $p$ is a non-root process)

(ND1) On changing the status to DETACHING:
    // *ask the parent to inherit all children*
    send ADD_CHILDREN$\langle children \rangle$ message to $parent$;

(ND2) On receiving ADD_CHILDREN$\langle newChildren \rangle$ message from process $q$:
    // *inherit children of the departing child*
    merge $newChildren$ with $children$;
    send OKAY_AC message to process $q$;

(ND3) On receiving OKAY_AC message from process $q$:
    // *instruct all children to update their parent*
    send UPDATE_PARENT$\langle parent \rangle$ message to all processes in $children$;

(ND4) On receiving UPDATE_PARENT$\langle newParent \rangle$ message from process $q$:
    // *update the parent*
    $parent := newParent$;
    send OKAY_UP message to process $q$;

(ND5) On receiving OKAY_UP message from process $q$:
        // *terminate the child relationship with* $q$
    remove $q$ from $children$;
    **if** (OKAY_UP message has been received from all processes in $children$) **then**
        // *ask the parent to terminate the child relationship*
        send REMOVE_CHILD message to $parent$;
        $parent := nil$;
    **endif**;

(ND6) On receiving REMOVE_CHILD message from process $q$:
    // *terminate the child relationship*
    remove $q$ from $children$;
    send OKAY_RC message to process $q$;

(ND7) On receiving OKAY_RC message from process $q$:
    // *no longer part of the spanning tree*
    $status$ := OUT;

Figure 6: Actions pertinent to the detaching phase of the control depart operation when the departing process is not a root.

process can safely modify the tree within its 1-hop neighborhood. Therefore, to establish that depart operation is live, it suffices to show that a process in trying phase is eventually able to obtain permission to depart from all its current neighbors. Specifically, we show that, given a system execution, if a process enters the trying phase at some point during the execution, then it eventually enters the detaching phase. Some of the notions we define for the liveness proof depend on the actual system execution. We assume the dependence to be implicit because we refer to the

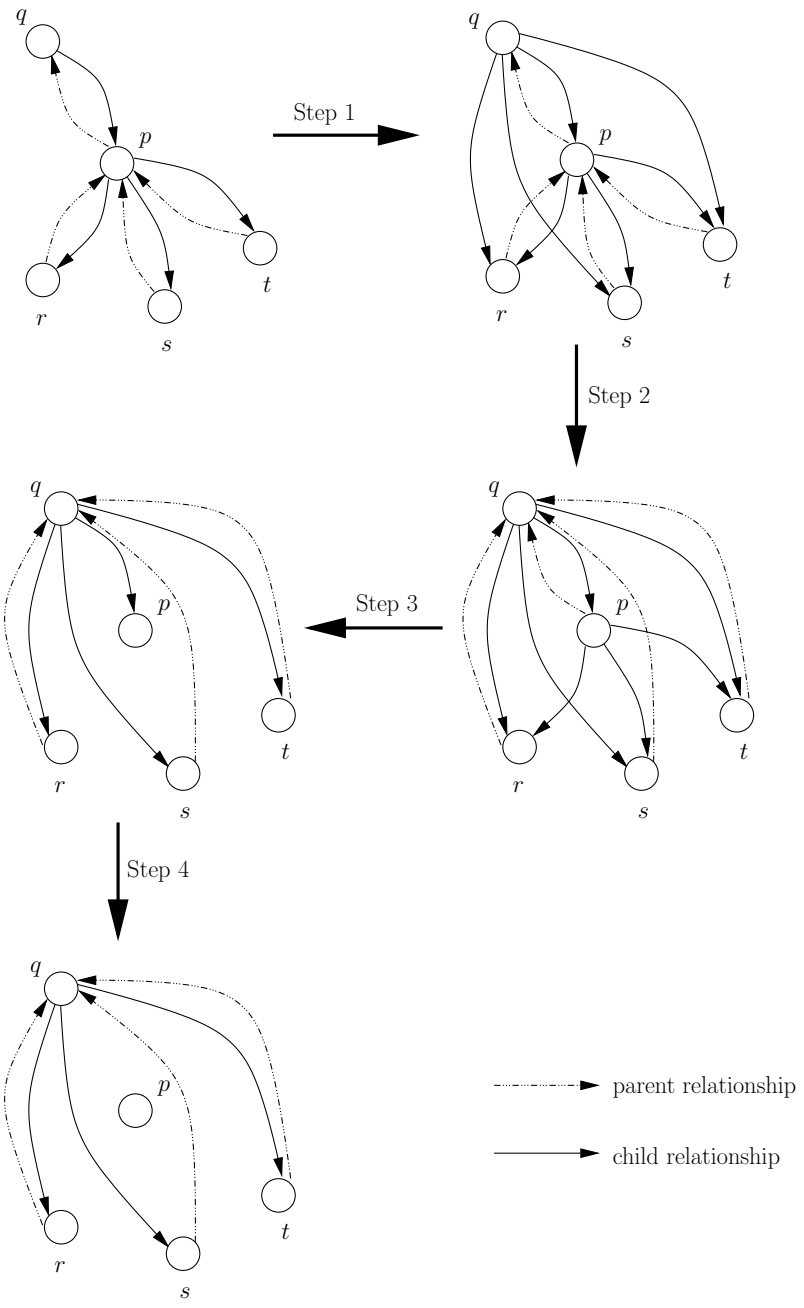


Figure 7: A step-by-step illustration of the detaching phase when the departing process is not the root.

same execution throughout the proof.

The main idea behind the liveness proof is to show that, if a departure request is not fulfilled for a sufficiently long time, then eventually the system reaches a state in which all current and future requests have higher timestamps than the given request. Once that happens, the request is

Actions pertinent to the detaching phase of the control depart operation for process $p$:
(Case 2: when $p$ is a root process)

*// Actions ND2 and ND4 in Figure 6 are also pertinent here*
*// They are not shown here because they can be used as is without any modification*

Additional variables:
$newRoot$: new root of the spanning tree;
$otherChildren$: set of children in the spanning tree except for the new root;

(RD1) On changing the status to DETACHING:
*// select one of the children to become the new root*
$newRoot :=$ some process in $children$;
$otherChildren := children \setminus \{newRoot\}$;
*// ask the new root to inherit all other children*
send ADD_CHILDREN$\langle otherChildren \rangle$ message to $newRoot$;

(RD2) On receiving OKAY_AC message from process $q$:
*// inherit children of the departing root*
send UPDATE_PARENT$\langle newRoot \rangle$ message to all processes in $otherChildren$;

(RD3) On receiving OKAY_UP message from process $q$:
*// terminate the child relationship with $q$*
remove $q$ from $children$;
**if** (OKAY_UP message has been received from all processes in $otherChildren$) **then**
*// instruct the selected child to become the new root*
send NEW_ROOT message to $newRoot$;
**endif**;

(RD4) On receiving NEW_ROOT message from process $q$:
*// become the new root*
$parent := p$;
send OKAY_NR message to process $q$;

(RD5) On receiving OKAY_NR message from process $q$:
*// terminate all relationships*
$parent := nil$;
$children := \emptyset$;
$status :=$ OUT;

Figure 8: Actions pertinent to the detaching phase of the control depart operation when the departing process is a root.

soon satisfied. We now provide a more formal proof of the liveness of the control depart operation.

We say that a process $p$ has an *extant* request for departure in a local state $x$, denoted $extant(x)$, if its status is either TRYING or DETACHING in $x$. Formally,

$$extant(x) \triangleq status(x) \in \{\text{TRYING}, \text{DETACHING}\} \quad (3)$$

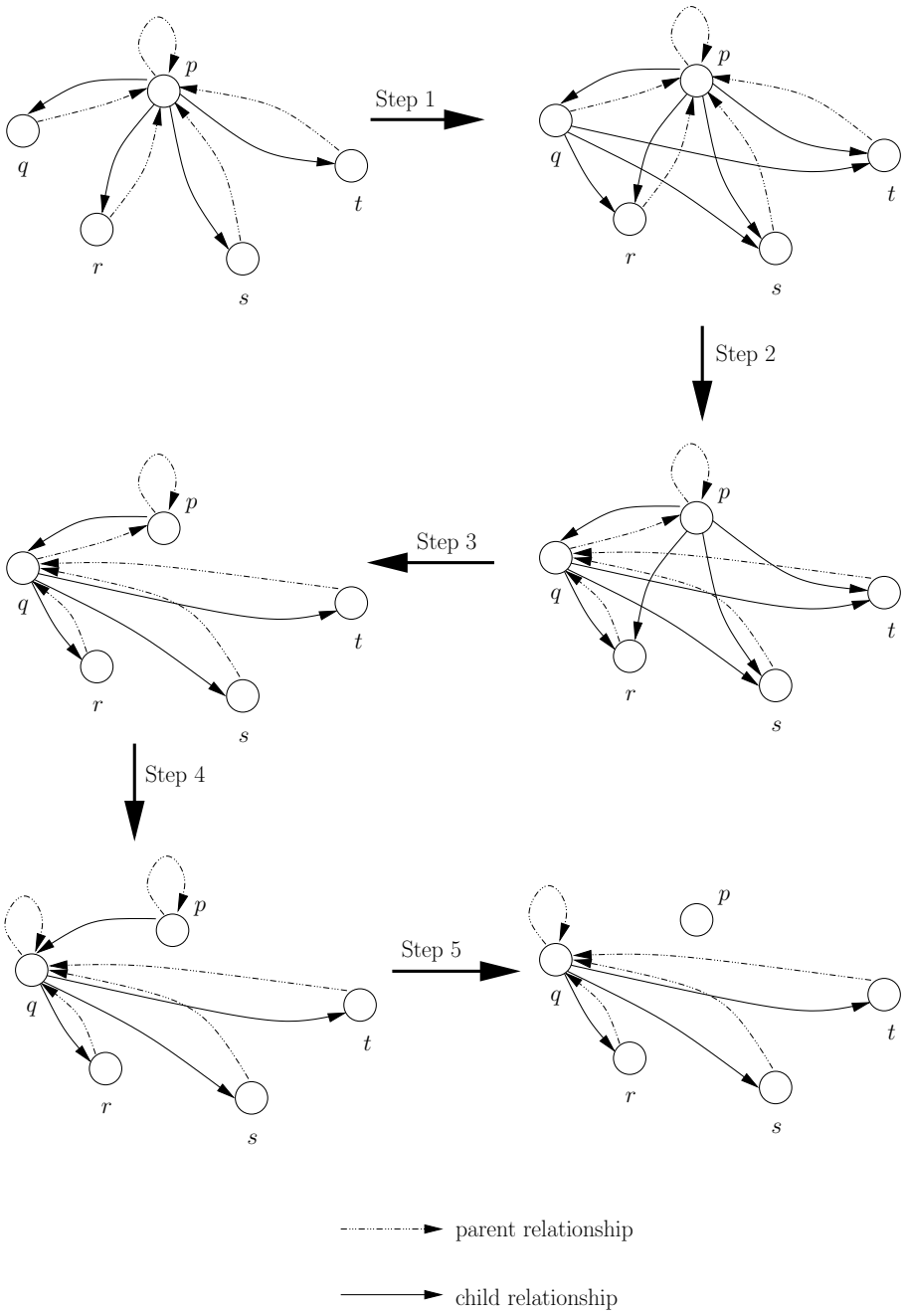


Figure 9: A step-by-step illustration of the detaching phase when the departing process is the root.

We use *extant-set*$(X)$ to denote the set of processes with extant requests in $X$. Formally,

$$\textit{extant-set}(X) \triangleq \{p \in \textit{process-set}(X) \mid \textit{extant}(X.p)\}$$

We say that an extant request has been satisfied or fulfilled once the request becomes non-extant. Given a process $p$ with extant request in some comprehensive state, we use *depart-ts*$(p)$ to refer to the timestamp of $p$'s request. For the liveness proof, we assume that a continuously

enabled event is eventually executed. Hereafter in this section, we assume a system execution to be implicit.

Given two comprehensive states $X$ and $Y$, we use $X \sqsubseteq Y$ to denote the fact that $Y$ is obtained from $X$ by executing zero or more events. Intuitively, if $X \sqsubseteq Y$ and $X \neq Y$, then $Y$ lies in the *future* of $X$. Further, we use $X \overset{k}{\sqsubseteq} Y$ denote the fact that $Y$ is obtained from $X$ by executing exactly $k$ events.

We define two notions involving extant departure requests: *potence* and *omnipotence*. Given a comprehensive state $X$, an extant request for departure by a process $p$ in $X$ is said to be *potent*, denoted $potent(p, X)$, if it has *smaller* timestamp than all requests for departure generated in the future of $X$. Formally,

$$\begin{array}{c} potent(p, X) \\ \triangleq \\ (p \in extant\text{-}set(X)) \wedge \\ \langle \forall\, Y : X \sqsubseteq Y : \langle \forall\, q : q \in extant\text{-}set(Y) \setminus extant\text{-}set(X) : depart\text{-}ts(p) < depart\text{-}ts(q) \rangle \rangle \end{array} \tag{4}$$

Intuitively, once a request becomes potent, it only has to compete with current requests for departure. Any future request for departure has a larger timestamp than it and, therefore, cannot prevent the requesting process from entering the detaching phase. A potent request by process $p$ in comprehensive state $X$ is said to be *omnipotent* if it has *smallest* timestamp among all extant requests for departure in $X$. Formally,

$$\begin{array}{rcl} omnipotent(p, X) & \triangleq & potent(p, X) \wedge \\ & & \langle \forall\, q : q \in extant\text{-}set(X) \setminus \{p\} : depart\text{-}ts(p) < depart\text{-}ts(q) \rangle \end{array} \tag{5}$$

It can be easily proved that:

**Proposition 2** *An omnipotent request is eventually satisfied. Formally,*

$$omnipotent(p, X) \quad \Rightarrow \quad \langle \exists\, Y : X \sqsubseteq Y : p \notin extant\text{-}set(Y) \rangle$$

Using Proposition 2, we show that:

**Lemma 3** *A potent request is eventually satisfied. Formally,*

$$potent(p, X) \quad \Rightarrow \quad \langle \exists\, Y : X \sqsubseteq Y : p \notin extant\text{-}set(Y) \rangle$$

| Notion | Meaning | Formal Definition |
|---|---|---|
| extant request | the requesting process has not yet departed from the system | (3) |
| potent request | the request has smaller timestamp than all future requests | (4) |
| omnipotent request | the request has smaller timestamp than all current and future requests | (5) |
| attached process | the process is currently attached or about to be attached to the spanning tree | (6) |
| relevant process | the process will execute at least one event in the future | (7) |
| local virtual time | current time of the process | (10) |
| global virtual time | current time of the system | (11) |

Table 3: Various notions used in proving the liveness of the depart operation.

**Proof:** Let $S$ denote the set of all processes with extant requests whose requests have a smaller timestamp than $p$'s request. The lemma can be proved by using induction on the size of $S$ and Proposition 2. □

From Lemma 3, it is sufficient to show that every departure request eventually either becomes potent or is satisfied. To that end, we define two notions involving processes. We say that a process $p$ is *attached* to the spanning tree in a comprehensive state $X$, denoted $attached(p, X)$, if (1) its status is ATTACHING and there is an ACCEPT message in transit towards it, or (2) its status is either IN, TRYING or DETACHING. Formally,

$$\begin{aligned} attached(p, X) \triangleq \Big( &(status(X.p) = \mathsf{ATTACHING}) \wedge \\ &(\text{there is an } \mathsf{ACCEPT} \text{ message in transit towards } p \text{ in } X) \Big) \bigvee \\ &(status(X.p) \in \{\mathsf{IN}, \mathsf{TRYING}, \mathsf{DETACHING}\}) \end{aligned} \tag{6}$$

We use $attached\text{-}set(X)$ to refer to the set of processes that are attached to the spanning tree in $X$. Formally,

$$attached\text{-}set(X) \triangleq \{p \in process\text{-}set(X) \mid attached(p, X)\}$$

Further, given a comprehensive state $X$, we say that a process $p$ is *relevant* in $X$, denoted $relevant(p, X)$, if it is attached to the spanning tree in $X$ and executes at least one event in the future of $X$. Formally,

$$relevant(p, X) \triangleq attached(p, X) \wedge \langle \exists\, Y : X \sqsubseteq Y : events(Y.p) \setminus events(X.p) \neq \emptyset \rangle \tag{7}$$

We define *relevant-set*$(X)$ to be the set of all processes that are relevant in $X$. Formally,

$$\textit{relevant-set}(X) \triangleq \{p \in \textit{process-set}(X) \mid \textit{relevant}(p, X)\}$$

Note that a *new* process can attach itself to the spanning tree only by contacting a process that is *already* part of the spanning tree. Therefore if no process in the spanning tree executes any event in the future, then no new process can ever join the spanning tree. Further, relevant set can only grow if a new process joins the spanning tree; otherwise it either shrinks or remains the same. The following properties can be easily verified:

$$\begin{aligned}(X \sqsubseteq Y) \wedge (\textit{relevant-set}(X) = \emptyset) \quad \Rightarrow \\ (\textit{attached-set}(X) = \textit{attached-set}(Y)) \wedge (\textit{relevant-set}(Y) = \emptyset)\end{aligned} \tag{8}$$

$$(X \sqsubseteq Y) \wedge (\textit{attached-set}(X) \supseteq \textit{attached-set}(Y)) \quad \Rightarrow \quad \textit{relevant-set}(X) \supseteq \textit{relevant-set}(Y) \tag{9}$$

We first show that *relevant-set*$(X)$ is non-empty as long as *extant-set*$(X)$ is non-empty.

**Lemma 4** *If a comprehensive state contains at least one process with extant request for departure, then it contains at least one process attached to the spanning tree that executes an event in the future. Formally,*

$$\textit{extant-set}(X) \neq \emptyset \quad \Rightarrow \quad \textit{relevant-set}(X) \neq \emptyset$$

**Proof:** Assume, on the contrary, that *extant-set*$(X) \neq \emptyset$ but *relevant-set*$(X) = \emptyset$. Consider the process in *extant-set*$(X)$ whose request has the smallest timestamp among all processes in *extant-set*$(X)$, say $p$. Clearly, $p$ is attached to the spanning tree in $X$, that is, $p \in$ *attached-set*$(X)$. Since *relevant-set*$(X) = \emptyset$, from (8), no request for departure is generated in the future. Therefore $p$'s request is potent in $X$. Further, the way $p$ has been chosen, its request is also omnipotent in $X$. From Proposition 2, $p$'s request is eventually satisfied, which, in turn, implies that $p$ eventually executes at least one event. □

Given a comprehensive state $X$ and a process $p$, we define *local virtual time* of $p$ in $X$, denoted

by $\text{lvt}(p, X)$, as follows:

$$\text{lvt}(p, X) \triangleq \begin{cases} \text{logical clock value of } p \text{ in } X.p & : \ status(X.p) \in \{\mathsf{IN}, \mathsf{TRYING}, \mathsf{DETACHING}\} \\ \text{logical timestamp of the } \mathsf{ACCEPT} \text{ message} & : \ (status(X.p) = \mathsf{ATTACHING}) \wedge \\ & \quad (\text{there is an } \mathsf{ACCEPT} \text{ message in transit towards } p \text{ in } X) \\ \text{undefined} & : \ \text{otherwise} \end{cases} \tag{10}$$

Note that the local virtual time of a process is monotonically non-decreasing. Based on the notion of local virtual time of a process, we define the notion of *global virtual time* of a system as follows:

$$\text{gvt}(X) \triangleq \begin{cases} \min\limits_{p \in relevant\text{-}set(X)} \{\text{lvt}(p, X)\} & : \ relevant\text{-}set(X) \neq \emptyset \\ \top & : \ \text{otherwise} \end{cases} \tag{11}$$

We assume that $\top$ is greater than any other value for virtual time. Note that a system execution may contain events executed by processes that are either still not attached to the spanning tree or were attached to the spanning tree before but are no longer attached to it. We do not include such processes in calculating global virtual time. We first show that global virtual time of a system is monotonically non-decreasing.

**Theorem 5** *Global virtual time of a system is monotonically non-decreasing. Formally,*

$$X \sqsubseteq Y \quad \Rightarrow \quad \text{gvt}(X) \leq \text{gvt}(Y)$$

**Proof:** In case $relevant\text{-}set(X) = \emptyset$, from (8), $relevant\text{-}set(Y) = \emptyset$. As a result, $\text{gvt}(X) = \text{gvt}(Y) = \top$ and the lemma holds trivially. Therefore assume that $relevant\text{-}set(X) \neq \emptyset$. Now, if $relevant\text{-}set(Y) = \emptyset$, then $\text{gvt}(Y) = \top$. Since $\text{gvt}(X) \neq \top$, $\text{gvt}(X) < \top$ and the lemma holds trivially. Therefore assume that $relevant\text{-}set(Y) \neq \emptyset$. In other words, $relevant\text{-}set(X) \neq \emptyset$ and $relevant\text{-}set(Y) \neq \emptyset$. Let $X = Z_0, Z_1, Z_2, \ldots, Z_m = Y$ be the sequence of comprehensive states starting from $X$ and ending at $Y$ such that $Z_i \overset{1}{\sqsubseteq} Z_{i+1}$ for each $i \in [0, m)$. Since $relevant\text{-}set(Y) \neq \emptyset$, from (8), $relevant\text{-}set(Z_i) \neq \emptyset$ for each $i \in [0, m]$. It is sufficient to show that $\text{gvt}(Z_i) \leq \text{gvt}(Z_{i+1})$ for each $i \in [0, m)$.

Let $Z_{i+1}$ be obtained from $Z_i$ by executing the event $e_i$. There are two cases to consider depending on the process to which $e_i$ belongs: $process(e_i) \notin attached\text{-}set(Z_i)$ and $process(e_i) \in attached\text{-}set(Z_i)$. In the first case, $attached\text{-}set(Z_i) = attached\text{-}set(Z_{i+1})$ and $relevant\text{-}set(Z_i) =$

*relevant-set*$(Z_{i+1})$. As a result, gvt$(Z_i)$ = gvt$(Z_{i+1})$. Now, consider the second case. There are two subcases to consider: *attached-set*$(Z_i) \supseteq$ *attached-set*$(Z_{i+1})$ and *attached-set*$(Z_i) \subsetneq$ *attached-set*$(Z_{i+1})$. In the first subcase, from (9), *relevant-set*$(Z_i) \supseteq$ *relevant-set*$(Z_{i+1})$. Since local virtual time of a process is monotonically non-decreasing, for each $p \in$ *relevant-set*$(Z_{i+1})$, lvt$(p, Z_{i+1}) \geq$ lvt$(p, Z_i)$. This implies that gvt$(Z_{i+1}) \geq$ gvt$(Z_i)$. Next, consider the second subcase. Let $p$ be the process on which $e_i$ is executed and let $q$ be the new process that is attached to the spanning tree in $Z_{i+1}$. Clearly, status of $q$ in $Z_{i+1}$ is ATTACHING and $p \in$ *relevant-set*$(Z_i)$. Therefore, by definition of local virtual time, lvt$(q, Z_{i+1})$ = lvt$(p, Z_{i+1}) >$ gvt$(Z_i)$. It can be verified that *relevant-set*$(Z_{i+1})$ is either *relevant-set*$(Z_i) \cup \{q\}$ or (*relevant-set*$(Z_i) \cup \{q\}) \setminus \{p\}$. In both cases, gvt$(Z_i) \leq$ gvt$(Z_{i+1})$ holds. □

We now show that if some departure request is not satisfied for a "long" time then global virtual time strictly increases.

**Lemma 6** *If there is extant request for departure, then eventually either the request is satisfied or global virtual time strictly increases. Formally,*

$$p \in \textit{extant-set}(X) \quad \Rightarrow \quad \langle \exists\, Y : X \sqsubseteq Y : (p \notin \textit{extant-set}(Y)) \vee (\text{gvt}(X) < \text{gvt}(Y)) \rangle$$

**Proof:** Consider a comprehensive state after $X$, say $Y$, in which all processes in *relevant-set*$(X)$ have executed at least one event since $X$. From the definition of relevant set, such a state exists. Assume that $p$'s request is still extant in $Y$, that is, $p \in$ *extant-set*$(Y)$. Therefore, from Lemma 4, *relevant-set*$(Y) \neq \emptyset$. We partition processes in *relevant-set*$(Y)$ into two categories: processes that belong *relevant-set*$(X)$ and processes that do not. We refer to the former as *old* processes and the latter as *new* processes.

First, consider an old process $q$, that is, $q \in$ *relevant-set*$(X) \cap$ *relevant-set*$(Y)$. From the way $Y$ has been chosen, lvt$(q, Y) >$ lvt$(q, X) \geq$ gvt$(X)$.

Next, consider a new process $q$, that is, $q \in$ *relevant-set*$(Y) \setminus$ *relevant-set*$(X)$. Let $V$ be the earliest comprehensive state in which $q$ becomes attached to the spanning tree and let $U$ denote the comprehensive state immediately preceding $V$. In other words, $X \sqsubseteq U \overset{1}{\sqsubseteq} V \sqsubseteq Y$, and $q \notin$ *attached-set*$(U)$ but $q \in$ *attached-set*$(V)$. Clearly, some process in *relevant-set*$(U)$, say $r$, receives a JOIN message from $q$ in $V$, accepts $q$ as a child, and sends an ACCEPT message to $q$. Clearly, $r \in$ *relevant-set*$(U)$. By definition of local virtual time of a joining process, we have,

$$\text{lvt}(q, V) = \text{lvt}(r, V)$$

$$\Rightarrow \quad \left\{ \begin{array}{l} V \text{ is obtained from } U \text{ by executing an event belonging to } r, \text{ which implies} \\ \text{that } \mathrm{lvt}(r, V) > \mathrm{lvt}(r, U) \end{array} \right\}$$

$\mathrm{lvt}(q, V) > \mathrm{lvt}(r, U)$

$\Rightarrow$ { since $r \in \mathit{relevant\text{-}set}(U)$, by definition of global virtual time, $\mathrm{lvt}(r, U) \geq \mathrm{gvt}(U)$ }

$\mathrm{lvt}(q, V) > \mathrm{gvt}(U)$

$\Rightarrow$ { global virtual time is monotonically non-decreasing }

$\mathrm{lvt}(q, V) > \mathrm{gvt}(X)$

$\Rightarrow$ { local virtual time is monotonically non-decreasing, therefore, $\mathrm{lvt}(q, Y) \geq \mathrm{lvt}(q, V)$ }

$\mathrm{lvt}(q, Y) > \mathrm{gvt}(X)$

Combining the two, for every process $q$ in $\mathit{relevant\text{-}set}(Y)$, $\mathrm{lvt}(q, Y) > \mathrm{gvt}(X)$. This implies that $\mathrm{gvt}(Y) > \mathrm{gvt}(X)$. □

We next show that if some departure request is not satisfied for a "long" time then it eventually becomes potent.

**Lemma 7** *If global virtual time has advanced beyond the timestamp of an extant request for departure, then the request has become potent. Formally,*

$$(p \in \mathit{extant\text{-}set}(X)) \wedge (\mathit{depart\text{-}ts}(p) < \mathrm{gvt}(X)) \quad \Rightarrow \quad \mathit{potent}(p, X)$$

**Proof:** If no departure request is generated in the future of $X$, then the lemma holds trivially. Therefore assume that at least one departure request is generated in the future of $X$. Suppose process $q$ generates a departure request in comprehensive state $Y$, where $X \sqsubseteq Y$. We have,

$(\mathit{depart\text{-}ts}(q) \geq \mathrm{gvt}(Y)) \wedge (X \sqsubseteq Y)$

$\Rightarrow$ { global virtual time is monotonically non-decreasing }

$\mathit{depart\text{-}ts}(q) \geq \mathrm{gvt}(X)$

$\Rightarrow$ { global virtual time has advanced beyond timestamp of $p$'s departure request }

$\mathit{depart\text{-}ts}(q) > \mathit{depart\text{-}ts}(p)$

This establishes the lemma. □

Finally, we can show that depart operation is live:

**Theorem 8** *Every request for departure is eventually satisfied. Formally,*

$$p \in \textit{extant-set}(X) \quad \Rightarrow \quad \langle \exists\, Y : X \sqsubseteq Y : p \notin \textit{extant-set}(Y) \rangle$$

**Proof:** Follows from Lemma 6, Lemma 7 and Lemma 3. □

Note that the liveness of our depart operation does not require the existence of a permanent process in the system. On the other hand, depart operation in [DMW05] is live only if at least one process in the ring is permanent. We next describe the snapshot algorithm.

# 5 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.

## 5.1 Achieving Consistency

To achieve consistency, we use Lai and Yang's approach for taking a consistent snapshot of a static distributed system [LY87]. 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. (Although not shown explicitly in formal descriptions, every message carries the instance number of the source process.) If a process receives a message with an instance number greater than its own, it first records its local state before delivering the message. A formal description of the actions necessary to maintain consistency is shown in Figure 10. It can be verified that:

**Theorem 9 (consistency)** *Two local states belonging to the same instance of the snapshot algorithm are consistent with each other.*

## 5.2 Achieving Completeness

We achieve completeness in two steps. We first describe how the current root of the spanning tree detect that its collection of local states has become complete. We then describe how to ensure that the collection at the root of the spanning tree eventually becomes complete.

Actions pertinent to ensuring consistency for process $p$:

Useful expressions:
$root \triangleq parent = p$

Additional variables:
$instance$: instance number of the latest snapshot algorithm participated so far;

// *For actions CS2 and CS3, MSG refers to any message (application as well as control)*

(CS1) On initiating a new instance of the snapshot algorithm:
// *only executed if* $p$ *is currently a root of the spanning tree, its status is either IN or*
// *TRYING and the previous instance of the snapshot algorithm has terminated*
$instance := instance + 1$;
call recordSnapshot( );

(CS2) Just before sending MSG to process $q$:
// *piggyback* $instance$ *on the message*
send MSG$\langle instance \rangle$ to process $q$;

(CS3) Just before receiving MSG$\langle nhbrInstance \rangle$:
**if** $(instance < nhbrInstance)$ **and** $(status \notin \{\text{OUT}, \text{ATTACHING}\})$ **then**
call recordSnapshot( );
**endif**;
$instance := \max\{instance, nhbrInstance\}$;
deliver MSG;

Figure 10: Actions pertinent to ensuring consistency.

### 5.2.1 Ensuring Safety

As explained earlier in Section 3, 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 3 ($f$-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,*

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

| Notion | Meaning | Formal Definition |
|---|---|---|
| active process | the process has joined the system and has neither left nor trying to leave the system | (1) |
| semi-active process | the process has joined the system but may be trying to leave the system | (2) |
| initiating process for a collection | the process that initiated the instance of the snapshot algorithm | - |
| largest root | the root has the largest rank in the current state | (12) |
| $f$-closed collection | the collection is closed with respect to the neighborhood function $f$ | Definition 3 |
| $f$-connected state | the local states of all processes that are currently semi-active are connected with respect to the neighborhood function $f$ | Definition 5 |

Table 4: Various notions used in proving the safety of the snapshot algorithm.

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 $f$. Given 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.

$$\rho(x) \quad \triangleq \quad \{parent(x) \mid parent(x) \neq nil\}$$

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 4 ($f$-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 process-set$(X)$. We say that there is an $f$-*path *from $p$ to $q$ in $X$, denoted by $f$-path$(p, q, X)$, if there exists a sequence of processes $s_i \in$ process-set$(X)$ 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 5 ($f$-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 semiactive-set$(X)$. Formally, $X$ is $f$-connected if*

$$\langle \forall p, q \in \textit{process-set}(X) : p \neq q : \{p, q\} \subseteq \textit{semiactive-set}(X) \;\Rightarrow\; f\textit{-path}(p, q, X) \rangle$$

Using the invariants (I1)–(I4), we show that every comprehensive state is actually $(\rho \cup \kappa)$-connected. To that end, we first prove an important property about the spanning tree maintained by our algorithm. Given a comprehensive state $X$, we construct a directed graph, denoted by $\rho\textit{-graph}(X)$, with the set of vertices as $\textit{process-set}(X)$ and the set of edges given by:

$$\{(p, \textit{parent}(X.p)) \mid p \in \textit{process-set}(X) \text{ and } \textit{parent}(X.p) \neq \textit{nil}\ \}$$

**Theorem 10** *The directed graph induced by parent variables of a comprehensive state is acyclic (except for self-loops).*

**Proof:** The proof is by induction on the number of non-initial events that have been executed to reach $X$, which is given by $|\textit{events}(X) \setminus \bot|$,.

**Base Case ($|\textit{events}(X) \setminus \bot| = 0$):** In this case, $X = \bot$. By assumption, $\rho\textit{-graph}(\bot)$ is acyclic.

**Induction Step ($|\textit{events}(X) \setminus \bot| = k + 1$):** Suppose $X$ is obtained from $W$ by executing the event $e$, that is, $W \overset{1}{\sqsubseteq} X$ and $e \in \textit{events}(X) \setminus \textit{events}(W)$. By induction hypothesis, $\rho\textit{-graph}(W)$ is acyclic. Let $e$ be an event on process $p$.

On executing $e$, a cycle may be formed in $\rho\textit{-graph}(X)$ only if $e$ causes the parent variable of $p$ to be changed such that: (1) $\textit{parent}(X.p) \neq \textit{parent}(W.p)$, and (2) $(\textit{parent}(X.p) \neq \textit{nil}) \wedge (\textit{parent}(X.p) \neq p)$. This may only happen if either $p$ is trying to join the system or some neighbor of $p$ is trying to depart from the system. Note that any cycle in $\rho\textit{-graph}(X)$, if it exists, should involve $p$; otherwise the same cycle also exists in $\rho\textit{-graph}(W)$—a contradiction. There two cases to consider:

- **Case 1 ($p$ is trying to join the system):** In this case, $\textit{status}(W.p) = \mathsf{ATTACHING}$, which implies that $p$ does not have any incoming edge in $\rho\textit{-graph}(W)$. This, in turn, means that $p$ cannot have any incoming edge in $\rho\textit{-graph}(X)$. As a result, $p$ cannot be involved in any cycle in $\rho\textit{-graph}(X)$.

- **Case 2 ($q$, the parent of $p$ in $W$, is trying to depart from the system):** In this case, $parent(W.p) = q$. Let $parent(X.p) = r$. In other words, the "old" parent of $p$ is $q$ and its "new" parent is $r$. There are two subcases to consider depending on whether $q$ is a root process in $W$.
  - **Case 2.1 ($q$ is not a root process in $W$):** In this case, $\rho\text{-}graph(X)$ contains the edge $(p, r)$ and $\rho\text{-}graph(W)$ contains the edges $(p, q)$ and $(q, r)$. Any cycle in $\rho\text{-}graph(X)$ involving $p$ should involve the edge $(p, r)$. This, in turn, implies that there is a cycle involving edges $(p, q)$ and $(q, r)$ in $\rho\text{-}graph(W)$—a contradiction.
  - **Case 2.2 ($q$ is a root process in $W$):** In this case, $\rho\text{-}graph(X)$ contains the edges $(p, r)$ and $(r, q)$ and $\rho\text{-}graph(W)$ contains the edge $(p, q)$. Any cycle in $\rho\text{-}graph(X)$ involving $p$ should involve the edges $(p, r)$ and $(r, q)$. This, in turn, implies that there is a cycle involving the edge $(p, q)$ in $\rho\text{-}graph(W)$—a contradiction.

This establishes the theorem. □

Next, we prove that every semi-active process in a comprehensive state has a $\rho$-path to the current root of the spanning tree. To capture the notion of "current" root of the spanning tree, we proceed as follows. Consider a comprehensive state $X$ and a process $p$ in $process\text{-}set(X)$ such that $root(X.p)$ holds. We say that $p$ has the largest rank among all root processes in $process\text{-}set(X)$, denoted by $largest(p, X)$, if $process\text{-}set(X)$ does not contain another root process with higher rank than $p$. Formally,

$$largest(p, X) \triangleq \langle \forall q \in process\text{-}set(X) : root(X.q) : rank(X.q) \leq rank(X.p) \rangle \qquad (12)$$

Clearly, if a root process has the largest rank among all root processes in a comprehensive state, then it is the current root of the spanning tree.

**Lemma 11** *Consider a comprehensive state $X$ and let $p$ be a process in $X$. Then,*

$$(parent(X.p) \neq nil) \wedge \neg root(X.p)$$

$$\Rightarrow$$

$$\langle \exists q \in process\text{-}set(X) : root(X.q) : \rho\text{-}path(p, q, X) \ \wedge \ largest(q, X) \rangle$$

**Proof:** Consider the *longest* $\rho$-path in $X$ starting from process $p$, say $\pi$, in which no process appears more than once. From Theorem 10, the path $\pi$ should terminate in a process $q$ such that

either $parent(X.q) = nil$ or $root(X.q)$. Assume that the antecedent holds. Clearly, $p \neq q$ which implies that the length of the path $\pi$ is at least two. Let the process in the path $\pi$ immediately before $q$ be $r$.

length of the path $\pi$ is at least two and no process appears more than once in $\pi$

$\Rightarrow$ { definition of $r$ }

$(r \text{ exists}) \wedge (r \neq q) \wedge (parent(X.r) = q)$

$\Rightarrow$ { using (I3) }

$(r \neq q) \wedge (parent(X.r) = q) \wedge (parent(X.q) \neq nil)$

$\Rightarrow$ { $q$ is the last process in $\pi$ }

$(r \neq q) \wedge (parent(X.r) = q) \wedge root(X.q)$

$\Rightarrow$ { using contrapositive of (I4) }

$root(X.q) \wedge largest(q, X)$

This establishes the lemma. □

Now, we identify two properties we use to prove the result. First, using (I3), for a comprehensive state $X$ and distinct processes $p$ and $q$ in $process\text{-}set(X)$,

$$\rho\text{-}path(p, q, X) \quad \Rightarrow \quad \kappa\text{-}path(q, p, X) \tag{13}$$

We refer to the above property as the *reversal property*. Second, using (I4), if a root process is semi-active, then it should have the largest rank among all root processes in a comprehensive state. Formally, for a comprehensive state $X$ and a process $p$ in $process\text{-}set(X)$,

$$root(X.p) \wedge (p \in semiactive\text{-}set(X)) \quad \Rightarrow \quad largest(p, X) \tag{14}$$

We are now ready to show that every comprehensive state is $(\rho \cup \kappa)$-connected.

**Theorem 12** *Every comprehensive state is $(\rho \cup \kappa)$-connected.*

**Proof:** Consider a comprehensive state $X$. In case $|semiactive\text{-}set(X)| \leq 1$, clearly, $X$ is $(\rho \cup \kappa)$-connected. Therefore assume that $|semiactive\text{-}set(X)| \geq 2|$. Consider two distinct processes $p$ and $q$ in $semiactive\text{-}set(X)$. First, consider the process $p$. We have,

$p \in semiactive\text{-}set(X)$

$$
\begin{aligned}
& \{ \text{ using (I1) } \} \\
& (p \in \textit{semiactive-set}(X)) \wedge (\textit{parent}(X.p) \neq \textit{nil}) \\
\equiv \quad & \{ \text{ predicate calculus } \} \\
& (p \in \textit{semiactive-set}(X)) \wedge (\textit{parent}(X.p) \neq \textit{nil}) \wedge (\textit{root}(X.p) \vee \neg \textit{root}(X.p)) \\
\Rightarrow \quad & \{ \text{ predicate calculus } \} \\
& \Big( (p \in \textit{semiactive-set}(X)) \wedge \textit{root}(X.p) \Big) \bigvee \Big( (\textit{parent}(X.p) \neq \textit{nil}) \wedge \neg \textit{root}(X.p) \Big) \\
\Rightarrow \quad & \{ \text{ using (14) } \} \\
& \Big( \textit{root}(X.p) \wedge \textit{largest}(p, X) \Big) \bigvee \Big( (\textit{parent}(X.p) \neq \textit{nil}) \wedge \neg \textit{root}(X.p) \Big) \\
\Rightarrow \quad & \{ \text{ using Lemma 11 } \} \\
& \Big( \textit{root}(X.p) \wedge \textit{largest}(p, X) \Big) \bigvee \\
& \Big( \langle \exists r \in \textit{process-set}(X) : \textit{root}(X.r) : \rho\textit{-path}(p, r, X) \wedge \textit{largest}(r, X) \rangle \Big)
\end{aligned}
$$

In other words, there exists a process $r$ in $\textit{process-set}(X)$, which may be same as $p$, such that:

$$\textit{root}(X.r) \wedge \textit{largest}(r, X) \wedge ((r = p) \vee \rho\textit{-path}(p, r, X))$$

Likewise, it can be shown that there exists a process $s$ in $\textit{process-set}(X)$, which may be same as $q$, such that:

$$\textit{root}(X.s) \wedge \textit{largest}(s, X) \wedge ((s = q) \vee \rho\textit{-path}(q, s, X))$$

Combining the two, it follows that both $\textit{largest}(r, X)$ and $\textit{largest}(s, X)$ hold. This implies that $r = s$ because two different root processes cannot have the same rank. Since $p \neq q$, either $p$ or $q$ is different from $r$. Without loss of generality, assume that $q \neq r$. We have,

$$
\begin{aligned}
& \textit{root}(X.r) \wedge \textit{largest}(r, X) \wedge \Big( (r = p) \vee \rho\textit{-path}(p, r, X) \Big) \wedge \rho\textit{-path}(q, r, X) \\
\Rightarrow \quad & \{ \text{ simplifying } \} \\
& \Big( (r = p) \vee \rho\textit{-path}(p, r, X) \Big) \wedge \rho\textit{-path}(q, r, X) \\
\Rightarrow \quad & \{ \text{ predicate calculus } \} \\
& \rho\textit{-path}(q, p, X) \vee \Big( \rho\textit{-path}(p, r, X) \wedge \rho\textit{-path}(q, r, X) \Big) \\
\Rightarrow \quad & \{ \text{ using (13) } \} \\
& \kappa\textit{-path}(p, q, X) \vee \Big( \rho\textit{-path}(p, r, X) \wedge \kappa\textit{-path}(r, q, X) \Big) \\
\Rightarrow \quad & \{ \text{ definition of } (\rho \cup \kappa)\text{-path } \} \\
& (\rho \cup \kappa)\textit{-path}(p, q, X) \vee \Big( (\rho \cup \kappa)\textit{-path}(p, r, X) \wedge (\rho \cup \kappa)\textit{-path}(r, q, X) \Big)
\end{aligned}
$$

$\Rightarrow$ { path concatenation }

$(\rho \cup \kappa)$-$path(p, q, X) \vee (\rho \cup \kappa)$-$path(p, q, X)$

$\Rightarrow$ { simplifying }

$(\rho \cup \kappa)$-$path(p, q, X)$

This establishes the theorem. □

We now provide a sufficient condition for a collection of local states to be complete.

**Theorem 13 ($f$-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) semiactive-set$(X) \neq \emptyset$, and (3) $CS(X)$ is $f$-connected, then $X$ is complete with respect to $CS(X)$.*

**Proof:** We have to show that *active-set*$(CS(X)) \subseteq$ *process-set*$(X)$. In case *active-set*$(CS(X)) = \emptyset$, trivially, *active-set*$(CS(X)) \subseteq$ *process-set*$(X)$. Therefore assume that *active-set*$(CS(X)) \neq \emptyset$. Consider some process in *active-set*$(CS(X))$. Also, let $q$ be some process in *semiactive-set*$(X)$. Note that $q$ exists because *semiactive-set*$(X) \neq \emptyset$. We have,

$(p \in$ *active-set*$(CS(X))) \wedge (q \in$ *semiactive-set*$(X))$

$\Rightarrow$ { $X \subseteq CS(X)$ implies *semiactive-set*$(X) \subseteq$ *semiactive-set*$(CS(X))$ }

$(p \in$ *active-set*$(CS(X))) \wedge (q \in$ *semiactive-set*$(CS(X)))$

$\Rightarrow$ { $CS(X)$ is $f$-connected }

$f$-$path(q, p, CS(X)) \wedge (q \in$ *process-set*$(X))$

$\Rightarrow$ { $X$ is $f$-closed }

$p \in$ *process-set*$(X)$

Equivalently, $p \in$ *active-set*$(CS(X)) \Rightarrow p \in$ *process-set*$(X)$. This implies that *active-set*$(CS(X))$ $\subseteq$ *process-set*$(X)$. □

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 $\kappa$-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. (Note that the snapshot algorithm described above does not satisfy liveness. However, at this point, we are only interested in the safety property, namely the collection is complete with respect to some comprehensive state. We later describe additions to the basic snapshot algorithm so as to ensure liveness.) From Theorem 12 and Theorem 13, it suffices to show that the collection returned by the snapshot algorithm is $(\rho \cup \kappa)$-closed. Clearly,

**Proposition 14** *The collection of local states returned by the snapshot algorithm is $\kappa$-closed.*

From our algorithm, only a root process can initiate the snapshot algorithm. Moreover, at the time of initiation, its status is IN or TRYING. Let $X$ denote the collection of local states returned by an instance of the snapshot algorithm. We use $initiator(X)$ to denote the process that initiated the instance of the snapshot algorithm that resulted in the collection $X$. Clearly,

$$root(CS(X).initiator(X)) \tag{15}$$

$$initiator(X) \in semiactive\text{-}set(X) \tag{16}$$

We next prove an important property about the initiating process.

**Lemma 15** *Consider a collection of local states $X$ returned by the snapshot algorithm. Then,*

$$p = initiator(X) \quad \Rightarrow \quad largest(p, CS(X))$$

**Proof:** For convenience, let $Y = CS(X)$. Assume, on the contrary, that $largest(p, Y)$ does not hold. Therefore there exists a process $q$ in $process\text{-}set(Y)$ such that $root(Y.q)$ holds and $rank(Y.p) < rank(Y.q)$. We have,

$root(Y.p) \wedge root(Y.q) \wedge (rank(Y.p) < rank(Y.q))$

$\Rightarrow$ { using (I4) }

$status(Y.p) = \mathsf{DETACHING}$

$\Rightarrow$ { definition of semi-active process }

$p \notin semiactive\text{-}set(Y)$

$\Rightarrow$ { $X \subseteq Y$ implies $semiactive\text{-}set(X) \subseteq semiactive\text{-}set(Y)$ }

$p \notin semiactive\text{-}set(X)$

$\Rightarrow$ { using (16) }

a contradiction

Therefore $largest(p, Y)$ holds. □

The next theorem establishes that the collection of local states returned by the snapshot algorithm is not only $\kappa$-closed but also $(\rho \cup \kappa)$-closed.

**Theorem 16** *The collection of local states returned by the snapshot algorithm is consistent and $(\rho \cup \kappa)$-closed.*

**Proof:** From Theorem 9 and Proposition 14, it suffices to prove that the collection is $\rho$-closed. Let the collection be denoted by $X$ and let $Y = CS(X)$. Consider a process $p$ in $process\text{-}set(X)$. From Proposition 14, $X$ is $\kappa$-closed. Therefore it suffices to prove that $X$ is $\rho$-closed as well. To that end, we have to show that $\rho(X.p) \subseteq process\text{-}set(X)$. This trivially holds if either $parent(X.p) = nil$ or $parent(X.p) = p$. Thus assume that $parent(X.p) \neq nil$ and $\neg root(X.p)$. Further, let $parent(X.p) = r$. We have,

$(parent(X.p) \neq nil) \wedge \neg root(X.p)$

{ $X \subseteq Y$ and $p \in process\text{-}set(X)$ implies $X.p = Y.p$ }

$(parent(Y.p) \neq nil) \wedge \neg root(Y.p)$

$\Rightarrow$ { using Lemma 11 }

$\langle \exists q \in process\text{-}set(Y) : root(Y.q) : \rho\text{-}path(p, q, Y) \wedge largest(q, Y) \rangle$

$\Rightarrow$ { using Lemma 15 }

$\langle \exists q \in process\text{-}set(Y) : root(Y.q) : \rho\text{-}path(p, q, Y) \wedge largest(q, Y) \rangle \wedge$

$root(Y.initiator(X)) \wedge largest(initiator(X), Y)$

$\Rightarrow$ { different root processes cannot have the same rank }

$\langle \exists q \in process\text{-}set(Y) : root(Y.q) : \rho\text{-}path(p, q, Y) \wedge largest(q, Y) \wedge (q = initiator(X)) \rangle$

$\Rightarrow$ { simplifying }

$\rho\text{-}path(p, initiator(X), Y)$

$\Rightarrow$ { definition of $r$ and $\rho$-path }

$\rho\text{-}path(r, initiator(X), Y) \vee (r = initiator(X))$

$\Rightarrow$ { using (13) }

$\kappa\text{-}path(initiator(X), r, Y) \vee (r = initiator(X))$

$\Rightarrow$ { $X$ is $\kappa$-closed and $initiator(X) \in process\text{-}set(X)$ }

$r \in process\text{-}set(X)$

| Notion | Meaning | Formal Definition |
|---|---|---|
| missing processes | set of processes whose local states are missing from the collection | (18) |
| pending processes | set of departing processes from which the process is supposed to receive the last collection | - |
| strictly attached process | the process has joined but not yet left the system | (21) |
| local instance number | latest instance of the snapshot algorithm that the process is currently aware of | - |
| local instance number at joining | local instance number of the process immediately after it joins the system | - |
| final children | set of children of the departing process when it enters the detaching phase | - |

Table 5: Various notions used in proving the liveness of the snapshot algorithm.

Therefore if $p \in \mathit{process\text{-}set}(X)$ then $\rho(X.p) \subseteq \mathit{process\text{-}set}(X)$. In other words, $X$ is $\rho$-closed. $\Box$

It follows from Theorem 13, Theorem 12, Theorem 16 and (16) that:

**Corollary 17 (safety)** *The collection of local states returned by the snapshot algorithm is (1) consistent and (2) complete with respect to some comprehensive state.*

### 5.2.2 Ensuring Liveness

The liveness of the snapshot algorithm is only guaranteed if the system eventually becomes *permanently quiescent* (that is, the set of processes does not change). Other algorithms for property detection make similar assumptions to achieve liveness [DIR97, DMW05]. Without this assumption, the spanning tree may continue to grow forever, thereby forcing the snapshot algorithm to collect local states of an ever increasing number of new processes. Specifically, new processes joining the system may prevent the snapshot algorithm from ever reaching the "bottom" of the spanning tree. Formally, a system becoming permanently quiescent in comprehensive state $X$, denoted $\mathit{perm\text{-}quiescent}(X)$, is defined as:

$$\mathit{perm\text{-}quiescent}(X) \triangleq \langle \forall\, Y : X \sqsubseteq Y : \langle \forall\, p : p \in \mathit{process\text{-}set}(Y) : \mathit{status}(Y.p) \in \{\mathsf{IN}, \mathsf{OUT}\rangle \,\rangle$$

Even with the above assumption, we still need to enhance/modify the spanning tree maintenance algorithm (specifically, the depart operation) and the basic snapshot algorithm to achieve liveness.

Modifications to the actions of process $p$ for the control depart operation:
// *Only actions that have to be modified are shown here*

Variables:
$pending$: set of previous neighbors who have left the spanning tree but their final snapshots have not been received yet; a process in trying phase can enter the detaching phase only after $pending$ has become empty;
$deferred$: set of children from which a REMOVE_CHILD message has been received but its processing has to be delayed

Useful expressions:
$inherited(q)$: the process from which child $q$ was inherited;

(T3') On receiving GRANT message from process $q$ or removing a process from $pending$:
**if** (GRANT message has been received from all processes in $neighbors$) **and** ($pending = \emptyset$) **then**
// *can depart safely*
$status$ := DETACHING;
**endif**;

(ND2') On receiving ADD_CHILDREN$\langle newChildren \rangle$ message from process $q$:
merge $newChildren$ with $children$;
// *remember that $q$ is departing*
add $q$ to $pending$;
send OKAY_AC message to process $q$;

(ND6') On receiving REMOVE_CHILD message from process $q$:
**if** ($inherited(q) \notin pending$) **then**
// *can allow $q$ to depart*
remove $q$ from $children$;
send OKAY_RC message to process $q$;
**else**
// *cannot allow to $q$ to depart as yet*
add $q$ to $deferred$;
**endif**;

(ND7') On receiving OKAY_RC message from process $q$:
call sendCollectedSnapshot( $q$ );
$status$ := OUT;

(RD5') On receiving OKAY_NR message from process $q$:
call sendCollectedSnapshot( $q$ );
$parent := nil$;
$children := \emptyset$;
$status$ := OUT;

Figure 11: Modifications to the control depart operation.

We make the following enhancement to the basic snapshot algorithm. Whenever a process records its local state, it sends a control message containing the current instance number to all its children instructing them to record their local states. Additionally, we make the following two modifications

```
Actions pertinent to ensuring completeness for process p:

Additional variables:
    collection: set of local states collected so far for the current instance of the snapshot algorithm;

(CO1) On invocation of recordSnapshot( ):
    // record local state
    collection := {current local state};
    send RECORD message to all processes in children;

(CO2) Whenever the contents of collection change:
    if (collection became κ-closed for the first time) then
        if root then evaluate property for collection;
        else
            sendCollectedSnapshot( parent );
        endif;
    endif;

(CO3) On receiving REPORT⟨nhbrCollection⟩ message from process q:
    if (collection and nhbrCollection are for the same instance) then
        merge nhbrCollection with collection;
    endif;
    if (q ∉ children) then
        // q is leaving the system and has sent its last collection
        remove q from pending, if present;
    endif;
    // deliver/process all deferred REMOVE_CHILD messages
    for each r ∈ deferred such that inherited(r) = q do
        remove r from children and deferred;
        send OKAY_RC message to process r;
    endfor;

(CO4) On invocation of sendCollectedSnapshot( q ):
    // send the set of local states collected so far to q
    if (q ≠ nil) then
        send REPORT⟨collection⟩ message to process q;
    endif;
```

Figure 12: Actions pertinent to ensuring completeness.

to the control depart operation. First, just before leaving the system, a process transfers its *latest* collection of local states, which may be empty, to another process even though the collection has not yet become $\kappa$-closed. If the process is departing as a non-root process, it transfers the collection to its parent. Otherwise, it transfers the collection to the new root. To implement this modification, before a process can make the transition from the trying phase to the detaching phase, it has to wait until it has received the collection from all those departing processes that are supposed to transfer their collection to it. Second, a process cannot leave the system until its older parent

has left the system and transfered its collection to the appropriate process. To implement this modification, whenever a process inherits new children, it remembers the process from which it inherited those children. It then prevents a child from leaving the system, by delaying processing of REMOVE_CHILD message received from it, until the process from which the child was inherited has transfered its (possibly incomplete) collection to it. The modifications made to the control depart operation are shown in Figure 11. A formal description of the complete snapshot algorithm is given in Figure 12.

It is possible that, even after a collection of local states at a process has become $\kappa$-closed, the process may receive a collection of local states for the same instance from another process (via a REPORT message). In case this happens, the process simply merges the collection it has received with its own collection. After merging, the new collection at the process may no longer be $\kappa$-closed. However, a process reports its collection to its parent for an instance only after it becomes $\kappa$-closed for the *first* time or the process is leaving the system. (Although we believe that the scenario described above cannot occur, we do not actually prove it.)

Observe that the enhancements/modifications described above neither violate the safety of the snapshot algorithm nor violate the liveness of the depart operation. At the worst, the modifications to the control depart operation may slightly prolong the trying and detaching phases of a process. The main idea behind the liveness proof is to show that, in spite of the dynamic nature of the spanning tree, any process whose local state is "missing" from the collection of a process that is still part of the system has to be a "child" of the latter. We now formally prove that the snapshot algorithm is live.

As in the liveness proof of the control depart operation, consider a system execution and assume that comprehensive states of the execution are ordered by $\sqsubseteq$. Note that a process, on receiving a message, may be forced to record its local state before delivering the message itself. To simplify the proof, we treat the two actions (recording the local state, if required, and delivering the message) as separate actions and, therefore, they correspond to *two separate events*. As a result of this assumption,

**Proposition 18** *When a process delivers a message, the instance number carried by the message is* less than or equal to *the instance number of the process at the time of the delivery.*

For a process $p$ in detaching phase, we define *final-children*$(p)$ to be the set of children of $p$ immediately after $p$ enters the detaching phase. Further, we refer to a child in *final-children*$(p)$

as a *final* child of $p$. Once a process enters the detaching phase, its set of children can only shrink. Therefore, given a local state $x$ of $p$,

$$status(x) = \mathsf{DETACHING} \quad \Rightarrow \quad children(x) \subseteq \textit{final-children}(p)) \tag{17}$$

The *local instance number* of a process $p$ in local state $x$, denoted $\mathrm{lin}(x)$, is the value of the instance number at $p$ in $x$. Consider a process $p$ and an instance $i$ of the snapshot algorithm for which $p$ records its local state. For a local state $x$ of $p$ with $\mathrm{lin}(x) = i$, let $collection(i, x)$ denote the collection of local states for instance $i$ in $x$. Further, let $missing(i, x)$ denote the set of processes whose local states are required to make $collection(i, x)$ $\kappa$-closed. Formally,

$$\begin{aligned} &missing(i, x) \triangleq \\ &\qquad \{q \mid \exists y \in collection(i, x) : q \in children(y) : q \notin \textit{process-set}(collection(i, x))\} \end{aligned} \tag{18}$$

When $i = 0$, we define $missing(i, x)$ to be empty. Let $pending(x)$ denote the set of departing processes that $p$ is waiting for in $x$ to transfer their latest collections to it. Since a process cannot enter the detaching phase until it has received collections from all such processes, we have,

$$status(x) = \mathsf{DETACHING} \quad \Rightarrow \quad pending(x) = \emptyset \tag{19}$$

Further, since a process prevents a child from leaving the system until the process from which the child was inherited has left the system, we have,

$$q \in pending(x) \quad \Rightarrow \quad \textit{final-children}(q) \ \subseteq \ (children(x) \cup \{p\}) \tag{20}$$

While a process is trying to join the spanning tree, it does not record any local state but its local instance number may continue to increase. Only after the status of a process becomes IN, that it starts participating in the snapshot algorithm. Let $x$ be a local state of a process $p$. We say that $p$ is *strictly attached* to the spanning tree in $x$, denoted by $\textit{strictly-attached}(x)$, if it status is IN, TRYING or DETACHING in $x$. Formally,

$$\textit{strictly-attached}(x) \ \triangleq \ status(x) \in \{\mathsf{IN}, \mathsf{TRYING}, \mathsf{DETACHING}\} \tag{21}$$

For a process $p$ that has already joined the system, we use $\mathrm{atjoining\text{-}lin}(p)$ to refer to its local instance number immediately after its status changes from ATTACHING to IN. The following lemma comes in useful for proving the liveness of the snapshot algorithm.

**Lemma 19** *Assume that process $q$ is a child of process $p$ when $p$ records its local state for instance $i$ of the snapshot algorithm. Then $q$ records its local state for instance $i$ or later of the snapshot algorithm while it is strictly attached to the spanning tree. Formally,*

$$(p \text{ records its local state for instance } i \text{ in } X) \ \wedge \ (q \in \mathit{children}(X.p)) \quad \Rightarrow$$
$$\langle \exists\, Y : X \sqsubseteq Y : \mathit{strictly\text{-}attached}(Y.q) \wedge (\mathrm{lin}(Y.q) \geq i) \rangle \ \wedge (i > \text{atjoining-lin}(q))$$

The proof has been moved to the appendix. The main idea is that, when a parent of a process records its local state, the process receives a RECORD message from either that parent or its next parent before possibly leaving the system. We now show that if a collection of local states at a process is missing local state of a departing process, then the last REPORT message sent by the latter to the former contains the missing local state.

**Lemma 20** *Consider comprehensive states $X$ and $Y$ with $X \stackrel{1}{\sqsubseteq} Y$, $e \in \mathit{events}(Y) \setminus \mathit{events}(X)$ and $i = \mathrm{lin}(X.p)$. Suppose, on executing event $e$, a process $p$ delivers the last REPORT message from a departing process $q$. Then the instance number carried by the last REPORT message is $i$. Formally,*

$$(\mathrm{lin}(X.p) = i) \ \wedge \ (q \in \mathit{missing}(i, X.p)) \ \wedge \ (q \in \mathit{pending}(X.p)) \ \wedge \ (q \notin \mathit{children}(X.p))$$
$$\Rightarrow$$
$$\textit{instance number carried by the last REPORT message from } q \textit{ is } i$$

**Proof:** If $q$ belongs to $\mathit{missing}(i, X.p)$, then the collection at $p$ in $X$ contains a local state $u$ such that $q \in \mathit{children}(u)$. Let $r = \mathit{process}(u)$ and $U$ be the comprehensive state in which $r$ records $u$—its local state for instance $i$. We have,

$$(r \text{ records its local state for instance } i \text{ in } U) \ \wedge \ (q \in \mathit{children}(U.r))$$

Using Lemma 19, there is a comprehensive state $V$ such that the local instance number of $q$ in $V$ is at least $i$ (that is, $\mathrm{lin}(V.q) \geq i$) and $q$ is strictly attached to the spanning tree in $V$ (that is, $\mathit{strictly\text{-}attached}(V.q)$). Let $W$ be the comprehensive state in which $q$ sends its last REPORT message to $p$. From Proposition 18, $\mathrm{lin}(W.q) \leq i$. However, $\mathrm{lin}(V.q) \leq \mathrm{lin}(W.q)$. We have, $i \leq \mathrm{lin}(V.q) \leq \mathrm{lin}(W.q) \leq i$, which implies that $\mathrm{lin}(V.q) = i$. □

It turns out that if the collection at a process is not yet $\kappa$-closed, then any local state missing from the collection has to belong to its "child": (1) current or departing if the status of the

process is IN or TRYING and (2) final if the status of the process is DETACHING. Formally, given a comprehensive state $X$,

$$(status(X.p) \in \{\mathsf{IN}, \mathsf{TRYING}\}) \wedge (i = \mathrm{lin}(X.p)) \quad \Rightarrow \quad missing(i, X.p) \subseteq (children(X.p) \cup pending(X.p)) \tag{22}$$

and

$$(status(X.p) = \mathsf{DETACHING}) \wedge (i = \mathrm{lin}(X.p)) \quad \Rightarrow \quad missing(i, X.p) \subseteq final\text{-}children(p) \tag{23}$$

We first prove that the above two properties are actually *invariants* maintained by both spanning tree maintenance and snapshot algorithms.

**Theorem 21** *Every comprehensive state satisfies properties (22) and (23).*

**Proof:** The proof is by induction on the number of non-initial events that have been executed to reach $X$, which is given by $|events(X) \setminus \bot|$. We show that the two properties hold after executing any event.

**Base Case ($|events(X) \setminus \bot| = 0$):** In this case, $X = \bot$. Clearly, $\bot$ trivially satisfies the two properties because $\mathrm{lin}(X.p) = 0$ for all processes $p \in process\text{-}set(X)$.

**Induction Step ($|events(X) \setminus \bot| = k+1$):** Suppose $X$ is obtained from $W$ by executing the event $e$, that is, $W \overset{1}{\sqsubseteq} X$ and $e \in events(X) \setminus events(W)$. By induction hypothesis, every comprehensive state $V$ with $V \sqsubseteq X$ satisfies properties (22) and (23). Let $e$ be an event on process $p$. There are several ways that the properties may be falsified on executing $e$:

- **Case 1 ($p$ records its local state for new instance):** Clearly, immediately after recording the local state, the collection at $p$ only consists of $p$'s state. Therefore if $i = \mathrm{lin}(X.p)$, then $missing(i, X.p) = children(X.p)$.
- **Case 2 (a process is removed from the set of children at $p$):** Let $q$ be the process removed from the set of children. There are two subcases to consider: $status(W.p) \in \{\mathsf{IN}, \mathsf{TRYING}\}$ or $status(W.p) = \mathsf{DETACHING}$. In the former subcase, $e$ involves delivering a REMOVE_CHILD message from $q$. However, $q$ still belongs to $pending(X.p)$. (Note that, after processing a REMOVE_CHILD message from $q$, $p$ may process deferred REMOVE_CHILD

messages from other processes. The two properties hold after each REMOVE_CHILD message is processed.) In the latter subcase, $e$ involves delivering an OKAY_UP message from $q$. This implies that $q \in \mathit{children}(W.p) \subseteq \mathit{final\text{-}children}(p)$.

- **Case 3 (a process is removed from the set of pending processes at $p$):** This case occurs when $p$ receives the last REPORT message from a departing process, say $q$. Clearly, $\mathit{children}(W.p) = \mathit{children}(X.p)$ and $\mathit{pending}(W.p) = \mathit{pending}(X.p) \cup \{q\}$. Let $q$ sent its last REPORT message to $p$ in comprehensive state $V$. From our assumption about events (namely, recording local state and delivering a message are separate events), $\mathrm{lin}(W.p) \geq \mathrm{lin}(V.q)$. If $q \notin \mathit{missing}(i, W.p)$ and $\mathrm{lin}(W.p) > \mathrm{lin}(V.q)$, then $X$ trivially satisfies the two properties. On the other hand, if $q \notin \mathit{missing}(i, W.p)$ and $\mathrm{lin}(W.p) = \mathrm{lin}(V.q)$, then, on executing $e$, the collection at $p$ is *merged* with the collection received from $q$. Likewise, if $q \in \mathit{missing}(i, W.p)$, then, from Lemma 20, $\mathrm{lin}(V.q) = \mathrm{lin}(W.p)$. This implies that, on executing $e$, the collection at $p$ is *merged* with the collection received from $q$. In both subcases, we have,

$$
\begin{array}{ll}
 & \mathit{missing}(i, X.p) \\
\subseteq & \{ \text{ collection in } X.p \text{ is union of collection in } W.p \text{ and } V.q \ \} \\
 & \mathit{missing}(i, W.p) \cup \mathit{missing}(i, V.q) \\
\subseteq & \{ \text{ induction step, that is, } W \text{ satisfies (22) and (23) } \} \\
 & \Big(\mathit{children}(W.p) \cup \mathit{pending}(W.p)\Big) \cup \mathit{final\text{-}children}(q) \\
\subseteq & \{ \text{ using (20) } \} \\
 & \mathit{children}(W.p) \cup \mathit{pending}(W.p) \cup \{p\} \\
\subseteq & \{ \text{ simplifying } \} \\
 & \mathit{children}(X.p) \cup \mathit{pending}(X.p) \cup \{p, q\}
\end{array}
$$

It can be easily verified that $\{p, q\} \cap \mathit{missing}(i, W.p) = \emptyset$. To show that $p \notin \mathit{missing}(i, W.p)$, note that if there exists a local state $x \in \mathit{collection}(i, W.p)$ such that $p \in \mathit{children}(x)$ then, using Lemma 19, $p$ learns about the instance $i$ of the snapshot algorithm after joining the system. This implies that $p$ indeed participates in instance $i$ of the snapshot algorithm and the collection at $p$ in $W.p$ contains a local state of $p$. Likewise, we can show that $q \notin \mathit{missing}(i, V.q)$. Thus $\mathit{missing}(i, X.p) \subseteq \mathit{children}(X.p) \cup \mathit{pending}(X.p)$.

- **Case 4 (status of $p$ changes):** There are two subcases to consider: status of $p$ changes from ATTACHING to IN or status of $p$ changes from TRYING to DETACHING. In the former

subcase, $\text{lin}(X.p) = 0$ implying that $X$ trivially satisfies (22) and (23). Therefore assume the latter subcase. The status of $p$ changes from TRYING to DETACHING either on delivering a GRANT message (signifying that $p$ has obtained permission to depart from all its neighbors) or on delivering a REPORT message from a departing process (after which the departing process is removed from the set of pending processes). If $e$ involves delivering a GRANT message, then $\mathit{children}(W.p) = \mathit{children}(X.p)$ and $\mathit{pending}(W.p) = \mathit{pending}(X.p) = \emptyset$. Clearly, $\mathit{missing}(i, X.p) = \mathit{missing}(i, W.p) \subseteq \mathit{children}(W.p) = \mathit{children}(X.p) = \mathit{final\text{-}children}(p)$. On the other hand, if $e$ involves delivering a REPORT message, then Case 3 applies.

- **Case 5 (collection at $p$ grows):** There are two subcases to consider: $p$ receives a collection from another process, say $q$, that may or may not be $\kappa$-closed. In both subcases, $\mathit{children}(W.p) = \mathit{children}(X.p)$ and $\mathit{pending}(W.p) = \mathit{pending}(X.p) \cup \{q\}$. In case $q$ is removed from $\mathit{pending}(W.p)$, Case 3 applies. Therefore assume that $\mathit{pending}(W.p) = \mathit{pending}(X.p)$. This implies that the collection received by $p$ is $\kappa$-closed, which implies that $\mathit{missing}(i, X.p) \subseteq \mathit{missing}(i, W.p) \subseteq \mathit{children}(W.p) \cup \mathit{pending}(W.p) = \mathit{children}(X.p) \cup \mathit{pending}(X.p)$.

This establishes the theorem. □

We next show that, as long as no new instance of the snapshot algorithm is initiated, every "pertinent" process records its local state for the instance currently in progress.

**Lemma 22** *Assume that a collection of local states at process $p$ for instance $i$ of the snapshot algorithm is missing a local state of process $q$. Then $q$ records its local state for instance $i$ or later of the snapshot algorithm while it is strictly attached to the spanning tree. Formally,*

$$q \in \mathit{missing}(i, X.p) \;\Rightarrow$$
$$\langle \exists\, Y : X \sqsubseteq Y : \mathit{strictly\text{-}attached}(Y.q) \wedge (\text{lin}(Y.q) \geq i) \rangle \;\wedge\; (i > \text{atjoining-lin}(q))$$

**Proof:** If $q$ belongs to $\mathit{missing}(i, X.p)$, then the collection at $p$ in $X$ contains a local state $u$ such that $q \in \mathit{children}(u)$. Let $r = \mathit{process}(u)$ and $U$ be the comprehensive state in which $r$ records $u$—the local state for instance $i$. We have,

$$(r \text{ records its local state for instance } i \text{ in } U) \;\wedge\; (q \in \mathit{children}(U.r))$$

Using Lemma 19, there is a comprehensive state $Y$ such that the local instance number of $q$ in $Y$ is at least $i$ and $q$ is still attached to the spanning tree in $Y$. □

Note that, when the collection at a process becomes $\kappa$-closed for the first time, the process sends the collection to its current parent in the spanning tree. Later, if its parent changes, it does not send the collection again to the new parent. We next show that once a process sends its $\kappa$-closed collection to its current parent, the collection is not "lost" even if its parent changes later. For a process $p$ and an instance $i$, we use $\mathit{reported\text{-}set}(i, p)$ to denote the collection of local states for instance $i$ that $p$ reports to its parent when the collection becomes $\kappa$-closed for the first time.

**Lemma 23** *Assume that a process $p$ has reported its collection for instance $i$ of the snapshot algorithm to its parent when it became $\kappa$-closed and no new instance of the snapshot algorithm is ever initiated. Then every parent of $p$ eventually receives the collection sent by $p$ while it still considers $p$ to be its child. Formally,*

$$(\mathrm{lin}(X.p) = i) \ \wedge \ (q = \mathit{parent}(X.p)) \ \wedge \ (p \text{ has already reported its collection for instance } i \text{ in } X)$$

$$\Rightarrow$$

$$\langle \exists Y : X \sqsubseteq Y : (p \in \mathit{children}(Y.q)) \wedge (\mathrm{lin}(Y.q) \geq i) \wedge (\mathrm{lin}(Y.q) = i \ \Rightarrow \ \mathit{reported\text{-}set}(i, p) \subseteq \mathit{collection}(i, Y.q)) \rangle$$

The proof has been moved to the appendix. The main idea is that, once a process sends its collection to its current parent in the spanning tree, any parent of the process (current or future) receives the collection before leaving the system and is, therefore, able to "pass it on" to the next parent. Assume that the system eventually becomes permanently quiescent. We now show that, as long as no new instance of the snapshot algorithm is initiated, the collection of local states for the instance currently in progress eventually becomes $\kappa$-closed at every process in the system.

**Theorem 24** *Assume that the system eventually becomes permanently quiescent. If no new instance of the snapshot algorithm is initiated, then the collection eventually becomes $\kappa$-closed at every process that participates in the current instance of the snapshot algorithm. Formally,*

$$\mathit{perm\text{-}quiescent}(X) \wedge (\mathit{missing}(i, X.p) \neq \emptyset) \ \Rightarrow \ \langle \exists \, Y : X \sqsubseteq Y : (\mathrm{lin}(Y.p) > i) \ \vee \ (\mathit{missing}(i, Y.p) = \emptyset) \rangle$$

**Proof:** Assume that the system has become permanently quiescent and no new instance of the snapshot algorithm is initiated after $i$. Even after the system has become permanently quiescent,

the pending set may still be non-empty for one or more processes. Since no more processes depart from the system hereafter, the system eventually reaches a comprehensive state after which the pending set becomes empty for all processes in the system and stays empty forever. Therefore, without loss of generality, assume that:

$$\mathit{perm\text{-}quiescent}(X) \wedge \langle \forall\, p : p \in \mathit{active\text{-}set}(X) : \mathit{pending}(X.p) = \emptyset \rangle$$

It suffices to prove that:

$$\mathit{missing}(i, X.p) \neq \emptyset \quad \Rightarrow \quad \langle \exists\, Y : X \sqsubseteq Y : \mathit{missing}(i, Y.p) = \emptyset \rangle$$

Note that, once the system becomes permanently quiescent, the spanning tree does not change thereafter. The property can be proved by induction on the height of a process in the spanning tree.

**Base Case ($p$ is a leaf process):** We have,

$$\begin{array}{ll} & \{ \text{ using Theorem 21 } \} \\ & \mathit{missing}(i, X.p) \subseteq \mathit{children}(X.p) \cup \mathit{pending}(X.p) \\ \Rightarrow & \{ \ \mathit{pending}(X.p) = \emptyset \ \} \\ & \mathit{missing}(i, X.p) \subseteq \mathit{children}(X.p) \\ \Rightarrow & \{ \text{ since } p \text{ is a leaf process, } \mathit{children}(X.p) = \emptyset \ \} \\ & \mathit{missing}(i, X.p) = \emptyset \end{array}$$

**Induction Step (the height of $p$ is $k+1$):** Assume that the property holds for any process whose height is at most $k$. Since the pending set has become empty for all processes and no process leaves the system, any collection a process receives via a REPORT message has to be $\kappa$-closed. This further implies that the missing set of a process (given by the set of processes whose local states are missing from the collection) can only *shrink*. Specifically, no new process can be added to the missing set of a process. Let $q$ be some process in $\mathit{missing}(i, X.p)$. Since $\mathit{pending}(X.p) = \emptyset$, from Theorem 21, $q \in \mathit{children}(X.p)$. From Lemma 22, $q$ eventually records its local state for instance $i$. By induction hypothesis, the collection at $q$ for instance $i$ eventually becomes $\kappa$-closed. Once that happens, $q$ sends its collection to its current parent. From Lemma 23, $p$ eventually receives $q$'s collection and merges it with its own. In other words, there exists a comprehensive state $U$ with $X \sqsubseteq U$ such that $q$'s collection is part of $p$'s collection in $U$. Clearly, $q$ no longer belongs to the

missing set of $p$ in $Y$. Repeating this argument for every process that belongs to the missing set of $p$, we obtain that the collection at $p$ for instance $i$ eventually becomes $\kappa$-closed.

This establishes the lemma. □

From Theorem 24, if the root initiates the next instance of the snapshot algorithm only after the current instance has terminated, then:

**Theorem 25 (liveness)** *Assuming that the system eventually becomes permanently quiescent, every instance of the snapshot algorithm terminates eventually.*

Note that, in practice, it is not necessary for the system to be permanently quiescent for the snapshot algorithm to terminate. Rather, it is sufficient for the system to stay quiescent for a "sufficiently long time" so as to allow the snapshot algorithm to reach the "bottom" of the spanning tree.

# 6 Complexity Analysis

Let $N_{\text{sys}}$ denote the number of processes that are ever part of the system before the property evaluates to true. Also, let $D_{\max}$ denote the maximum degree attained by any process in the spanning tree. Note that the number of processes that are ever neighbors of a given process may actually be much larger than $D_{\max}$.

We first analyze the message complexity of the snapshot algorithm. Our snapshot algorithm exchanges two types of messages, namely RECORD and REPORT. We show that each process is involved in only a small number of RECORD messages per instance of the snapshot algorithm.

**Lemma 26** *A process receives at most two RECORD messages for every instance of the snapshot algorithm.*

**Proof:** Note that a process sends a RECORD message only when it records its local state for a *new* instance of the snapshot algorithm, and, moreover, it sends a RECORD message only to its children at the time of the recording. As a result, a process can receive a RECORD message only from its parent. We claim that a process can receive at most two RECORD messages for each instance of the snapshot algorithm. Consider a process $p$ and an instance $i$ of the snapshot algorithm such that at least one parent of $p$ sends a RECORD message to $p$ for instance $i$. Let $q$ be the *earliest* parent

of $p$ that sends a RECORD message to $p$ for instance $i$. In case $q$ is the last parent of $p$, clearly, no other process sends a RECORD message for instance $i$ to $p$. Therefore assume that $q$ is not the last parent of $p$. If $q$ records its local state for instance $i$ before sending the ADD_CHILDREN message to the next parent of $p$, then any new parent of $p$ records its local state for instance $i$ *before* inheriting $p$. This implies that no new parent of $p$ (after $q$) sends a RECORD message for instance $i$ to $p$. Therefore assume that $q$ records its local state for instance $i$ after sending the ADD_CHILDREN message to the next parent of $p$, say $r$. It can be proved that $r$ records its local state for instance $i$ before delivering the last REPORT message from $q$. Specifically, $r$ records its local state for instance $i$ before sending any ADD_CHILDREN message (if at all). This ensures that any new parent of $p$ after $r$ records its local state for instance $i$ before inheriting $p$ and, as a result, $p$ does not receive any more RECORD messages for instance $i$. □

Also, note that every process sends at most two REPORT messages for each instance of the snapshot algorithm (when its collection becomes $\kappa$-closed for the first time and when it is leaving the system). Therefore the message complexity of a single instance of the snapshot algorithm is given by $O(N_{\text{sys}})$.

We next analyze the message complexity of the control depart operation. Clearly, the message complexity of the detaching phase per departing process is given by $O(D_{\max})$. It remains to bound the message complexity of trying phase. We refer to a REQUEST message sent by a process to its neighbor as successful if the former receives a GRANT message from the latter; otherwise it is unsuccessful. We charge a successful REQUEST message to the sender of the message and an unsuccessful REQUEST message to the receiver of the message. Clearly, the number of successful REQUEST message per departing process is bounded by $O(D_{\max})$. It can be verified that the number of unsuccessful REQUEST messages that can be charged to any departing process is also bounded by $O(D_{\max})$. This implies that the message complexity of the trying phase per departing process is also given by $O(D_{\max})$.

# 7 Discussion

Baldoni *et al.* [BHP04] describe another spanning tree maintenance algorithm with focus on maintaining group connectivity in a dynamic environment. Our focus, on the other hand, is on collecting a snapshot of the system which can be used to evaluate a stable property in a meaningful manner.

To that end, we design a control depart operation that *does not interfere* with an ongoing snapshot algorithm. Further, our control depart operation, which uses logical timestamp to break ties, is general in the sense that it can be used with any structure and not just a spanning tree. (In fact, the proof of liveness of the control depart operation carries over to any structure.) In Baldoni *et al.*'s algorithm [BHP04], on the other hand, to avoid neighboring nodes from departing concurrently, ties are broken using parent-child relationship. Specifically, parent has preference over its child if both of them are trying to depart at the same time. Breaking time using parent-child relationship complicates departure of the root especially if new roots themselves wish to depart from the system.

Note that, to ensure liveness of our control join operation, the system should contain at least one permanent process [BHP04]. In case there is no permanent process in the system, our spanning tree maintenance and snapshot algorithms still work correctly in the sense that safety of our detection algorithm is never violated. The assumption about permanent processes made by Dhamdhere *et al.* [DIR97], on the other hand, is necessary to ensure the correctness (even safety) of the detection algorithm.

Our spanning tree maintenance algorithm has several undesirable characteristics. First, our control depart operation has relatively high *worst-case* time-complexity in the sense that 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 detaching phase. Second, when child of a process leaves, it inherits all children of the departing child. As a result, if several children of a process leave the system, then the process may inherit a large number of children, thereby increasing its degree in the spanning tree. This, in turn, increases the cost of departure of the process (in terms of the number of messages exchanged) if it decides to leave the system later. Note that there is a trade off between the degree of a process in the spanning tree and the height of the spanning tree. Having lower degree decreases the cost of the depart operation—in terms of number of messages—but increases the time it takes for an instance of the snapshot algorithm to terminate by indirectly increasing the height of the tree. An interesting research problem is to develop a depart operation that minimizes the amount of time a process has to stay in the system once the process has decided to leave the system. Further, it minimizes the number of control messages that are exchanged during departure while preventing the tree from becoming too long.

In this paper, we assume that, before leaving the system, a process executes a special depart operation to ensure that the tree remains connected even after the process has departed. Li *et al.* refer to this type of departure as *active* departure [LMP04]. In *passive* departure, on the

other hand, a process simply leaves without executing any special depart operation [LMP04]. As a result, the tree may be temporarily disconnected. The remaining process in the system then have to "discover" each other eventually to regain connectivity. While passive departure simplifies the departure of a process, it becomes it more difficult to design a correct property detection algorithm. An advantage of passive departure is that it is not necessary to distinguish between departure of a process and failure of a process because process failure can be treated as process departure. processes are reliable and never fail. Another interesting research problem is to design a stable property detection algorithm when processes are prone to failures (*e.g.*, by crashing).

## References


[AMG03] R. Atreya, N. Mittal, and V. K. Garg. Detecting Locally Stable Predicates without Modifying Application Messages. In *Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS)*, pages 20–33, La Martinique, France, December 2003.

[AV94] S. Alagar and S. Venkatesan. An Optimal Algorithm for Recording Snapshots using Casual Message Delivery. *Information Processing Letters (IPL)*, 50:311–316, 1994.

[BHP04] R. Baldoni, J. Hélary, and S. T. Piergiovanni. Maintaining Group Connectivity in Dynamic Asynchronous Distributed Systems. Technical report, Dipartimento di Informatica e Sistemistica "A.Ruberti", Università di Roma la Sapienza, 2004.

[CL85] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM84] K. M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984.

[CMH83] K. M. Chandy, J. Misra, and L. M. Haas. Distributed Deadlock Detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[DIR97] D. M. Dhamdhere, S. R. Iyer, and E. K. K. Reddy. Distributed Termination Detection for Dynamic Systems. *Parallel Computing*, 22:2025–2045, 1997.

[dis] distributed.net. http://www.distributed.net/.

[DMW05] D. Darling, J. Mayo, and X. Wang. Stable Predicate Detection in Dynamic Systems. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, 2005.

[DS80] E. W. Dijkstra and C. S. Scholten. Termination Detection for Diffusing Computations. *Information Processing Letters (IPL)*, 11(1):1–4, 1980.

[Fra80] N. Francez. Distributed Termination. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):42–55, January 1980.

[HR82] G. S. Ho and C. V. Ramamoorthy. Protocols for Deadlock Detection in Distributed Database Systems. *IEEE Transactions on Software Engineering*, 8(6):554–557, November 1982.

[Lai86] T. H. Lai. Termination Detection for Dynamic Distributed Systems with Non-First-In-First-Out Communication. *Journal of Parallel and Distributed Computing (JPDC)*, 3:577–599, 1986.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[LMP04] X. Li, J. Misra, and C. G. Plaxton. Active and Concurrent Topology Maintenance. In *Proceedings of the Symposium on Distributed Computing (DISC)*, pages 320–334, 2004.

[LY87] T.-H. Lai and T. H. Yang. On Distributed Snapshots. *Information Processing Letters (IPL)*, 25(3):153–158, May 1987.

[MS94] K. Marzullo and L. Sabel. Efficient Detection of a Class of Stable Properties. *Distributed Computing (DC)*, 8(2):81–91, 1994.

[RA81] G. Ricart and A. K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM (CACM)*, 24(1):9–17, January 1981.

[SS94] A. Schiper and A. Sandoz. Strong Stable Properties in Distributed Systems. *Distributed Computing (DC)*, 8(2):93–103, 1994.

[WM04] X. Wang and J. Mayo. 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, April 2004.

# A Omitted Proofs

**Proof for Lemma 19:** We first show that the local instance number of $q$ eventually advances to at least $i$ before $q$ leaves the system, if at all. A process, on recording its local state, sends a record message to all its children. Therefore $p$ sends a RECORD message carrying instance number $i$ to $q$ in $X$. If $q$ never leaves the system, then it eventually receives the RECORD message sent by $p$ carrying instance number $i$. Therefore assume that $q$ eventually leaves the system. There are two cases to consider:

**Case 1 ($p$ either never leaves the system or leaves the system after $q$):** In this case, $p$ receives a REMOVE_CHILD message from $q$ asking it to remove $q$ from its set of children. On processing the REMOVE_CHILD message, $p$ sends an OKAY_RC message to $q$. Note that, when $p$ records its local state for instance $i$, it considers $q$ to be one of its children. This implies that $p$ sends a RECORD message carrying instance number $i$ to $q$ before it sends the OKAY_RC message to $q$. Since messages are delivered in the order in which they are sent, $q$ receives the RECORD message carrying instance number $i$ from $p$ before it receives the OKAY_RC message from $p$.

**Case 2 ($p$ leaves the system before $q$):** In this case, just before $p$ enters the detaching phase, $q$ considers $p$ to be its parent. If $p$ records the local state for instance $i$ before sending an UPDATE_PARENT message to $q$, then $q$ receives the RECORD message for instance $i$ before receiving the UPDATE_PARENT message from $p$. Therefore assume that $p$ records the local state for instance $i$ after sending an UPDATE_PARENT message to $q$. Note that $p$ cannot be a root process because a root process initiates a new instance of the snapshot algorithm only when it is semi-active, that is, its status is IN or TRYING. This implies that a root process cannot record its local state once it enters the detaching phase. Let $r$ be the parent of $p$ when $p$ is detaching itself from the spanning tree and $W$ be the comprehensive state immediately before $r$ delivers the last report message from $p$. Our algorithm ensures that $p \in \mathit{pending}(W.r)$. Also, from (20), $q \in \mathit{children}(W.r)$. Further, from Proposition 18, $\mathrm{lin}(W.r) \geq i$. Again, there are two subcases to consider: $r$ either (1) never leaves the system or leaves the system after $q$ or (2) $r$ leaves the system before $q$. Note that the two subcases (which involve $r$ and $q$) are similar to the two cases (which involve $p$ and $q$). However, unlike before, we are guaranteed that $r$ records its local state for instance $i$ before entering the detaching phase, if applicable. In the first subcase, the local instance number of $q$ advances to at least $i$ when $q$ receives an OKAY_RC message from $r$. In the second subcase, the local instance

number of $q$ advances to at least $i$ when $q$ receives an UPDATE_PARENT message from $r$.

It now suffices to show that the local instance number of $q$ stays smaller than $i$ until $q$ actually joins the spanning tree and its status becomes IN. Let $r$ denote the process that $q$ attaches to when $q$ joins the spanning tree. Immediately after $r$ accepts $q$ as its child, the local instance number of $r$ should be strictly less than $i$. Assume the contrary. It can be proved that any process that inherits $q$ later has to record its local state for instance $i$ before inheriting $q$ (to ensure consistency). This contradicts the fact $q$ is a child of $p$ when $p$ records its local state for instance $i$. It follows therefore that, immediately after $q$ delivers the ACCEPT message from $r$, the local instance number of $q$ is strictly less than $i$. □

**Proof for Lemma 23:** Consider a process $q$ that becomes a parent of $p$ after the collection of local states at $p$ for instance $i$ has become $\kappa$-closed. We define *age* of $q$, denoted $age(q)$, as the number of times parent variable at $p$ has changed since the collection became $\kappa$-closed. The proof is by induction on age of parent process.

**Base Case ($age(q) = 0$):** In this case, $q$ is a parent of $p$ when the collection at $p$ becomes $\kappa$-closed. Further, from (I3), $p \in children(X.q)$, that is, $children(X.q) \neq \emptyset$. From (I2), $status(X.q) \in \{\mathsf{IN}, \mathsf{TRYING}, \mathsf{DETACHING}\}$, that is, $q$ is strictly attached to the spanning tree in $X$. If $q$ never leaves the system, then, clearly, it receives the collection sent by $p$. Therefore assume that $q$ leaves the system eventually. There are two cases to consider:

- **Case 1 ($p$ never leaves the system or leaves the system after $q$):** In this case, $q$ receives the collection before it receives the OKAY_UP message from $p$.
- **Case 2 ($p$ leaves the system before $q$):** Note that $p$ sends the REMOVE_CHILD message to $q$ after $X$. This is because, when a process sends REMOVE_CHILD message, its parent variable is *nil* but $parent(X.p) \neq nil$ . Since messages are received in the order in which they are sent, $q$ receives the collection before it receives the REMOVE_CHILD message from $p$.

In both cases, $q$ receives the collection sent by $p$ before $q$ possibly leaves the system. It can be verified that the lemma holds immediately after $q$ delivers the REPORT message carrying the collection from $p$.

**Induction Step ($age(q) = k+1$):** Consider the previous parent $r$ with $age(r) = k$. Let $X_k$ be a comprehensive state such that $r = parent(X_k.p)$. If $\text{lin}(X_k.p) > i$, then, the local instance number of $q$ advances beyond $i$ once $q$ receives REMOVE_CHILD message from $r$. Therefore assume that $\text{lin}(X_k.p) = i$. We have,

$$(\text{lin}(X_k.p) = i) \wedge (r = parent(X_k.p)) \wedge (p \text{ has already sent its collection for instance } i \text{ in } X_k)$$

If the local instance number of $r$ advances beyond $i$ before $r$ leaves the system, then the local instance number of $q$ advances beyond $i$ once $q$ receives the last REPORT message from $r$. As a result, the lemma trivially holds. Therefore assume that the local instance number of $r$ stays less than or equal to $i$ while $r$ is part of the system. Using induction hypothesis, there exists a comprehensive state $Y_k$ such that:

$$(\text{lin}(Y_k.r) = i) \wedge (reported\text{-}set(i,p) \subseteq collection(i, Y_k.r)) \wedge (p \in children(Y_k.r))$$

Note that $r$ sends the REMOVE_CHILD message to $q$ after $Y_k$. This is because, when a process sends REMOVE_CHILD message, it does not have any children but $p \in children(Y_k.r)$. Clearly, $q$ receives the collection of $p$ for instance $i$ as part of the last REPORT message sent by $r$ while $q$ is strictly attached to the spanning tree. It can be verified that the lemma holds immediately after $q$ delivers the last REPORT message sent by $r$. □