Nonintrusive Snapshots Using Thin Slices

Ajay D. Kshemkalyani and Bin Wu

Computer Science Department, Univ. of Illinois at Chicago, Chicago, IL 60607, USA
{ajayk, bwu}@cs.uic.edu

Abstract. This paper gives an efficient algorithm for recording consistent snapshots of an asynchronous distributed system execution. The nonintrusive algorithm requires $6(n-1)$ control messages, where n is the number of processes. The algorithm has the following properties. (P1) The application messages do not require any changes, not even the use of timestamps. (P2) The application program requires no changes, and in particular, no inhibition is required. (P3) Any process can initiate the snapshot. (P4) The algorithm does not use the message history. A simple and elegant three-phase strategy of uncoordinated observation of local states is used to give a consistent distributed snapshot. Two versions of the algorithm are presented. The first version records consistent process states without requiring FIFO channels. The second version records process states and channel states consistently but requires FIFO channels. The algorithm also gives an efficient way to detect any stable property, which was an unsolved problem under assumptions (P1)-(P4).

1 Problem Definition

A distributed system is modeled as a directed graph (N, L), where N is the set of processes and L is the set of links connecting the processes. Let $n = |N|$ and $l = |L|$. A distributed snapshot represents a consistent global state of the system. Recording distributed snapshots of an execution is a fundamental problem in asynchronous message-passing systems. Since the seminal algorithm of Chandy and Lamport [3] which is a non-inhibitory algorithm that requires FIFO channels and $2l$ messages to record a snapshot, plus additional messages to assemble the snapshot, several algorithms have been proposed. A survey is given in [8].

This paper gives an efficient nonintrusive non-inhibitory algorithm for recording consistent snapshots of an asynchronous distributed system execution. The algorithm requires $6(n-1)$ control messages and has the following properties.

P1. The application messages require no changes, not even timestamps.
P2. The application program requires no changes, implying no inhibition.
P3. Any process can initiate the snapshot.
P4. The algorithm does not require the log of the message history.

These properties are important for ubiquitous and pervasive computing. A simple and elegant three-phase strategy of uncoordinated observation of local states gives a consistent distributed snapshot. Two versions of the algorithm are

Table 1. Comparison of global snapshot algorithms. The acronym p.b. denotes that control information is piggybacked on the application messages.

Algorithm	Channels required	Non-inhibitory	Application messages unmodified	Number of control messages	Snapshot collection not needed	Message history not used
Chandy-Lamport [3]	FIFO	Y	Y	$O(n^2)$	N	Y
Spezialetti-Kearns [16]	FIFO	Y	Y	$O(n^2)$	N	Y
Venkatesan [17]	FIFO	Y	Y	$O(n^2)$	N	Y
Helary [5] (wave sync.)	FIFO	N	Y	$O(n^2)$	N	Y
Helary [5] (wave sync.)	non-FIFO	N	Y	$O(n^2)$	N	Y
Lai-Yang [10]	non-FIFO	Y	N (p.b.)	$O(n^2)$	N	N
LRV [12]	non-FIFO	Y	N (p.b.)	$O(n^2)$	N	N
Mattern [14]	non-FIFO	Y	N (p.b.)	$O(n)$	N	Y
Acharya-Badrinath [1]	CO	Y	Y	$2n$	Y	Y
Alagar-Venkatesan [2]	CO	Y	Y	$3n$	Y	Y
Proposed snapshot	FIFO	Y	Y	$6n$	Y	Y
Proposed snapshot (w/o channel states)	non-FIFO	Y	Y	$6n$	Y	Y

presented: the first version records consistent process states without requiring FIFO channels and without using any form of message send/receive or event counters, whereas the second version records process states and channel states consistently but requires FIFO channels. Critchlow and Taylor have shown that for a system with non-FIFO channels, a snapshot algorithm must either use piggybacking or use inhibition [4]. Hence, the second version of the algorithm cannot be improved upon to also record channel states while retaining the properties of no inhibition and no piggybacking while using non-FIFO channels.

Table 1 compares the proposed algorithms with the existing algorithms. Besides serving as a general-purpose snapshot algorithm, the proposed algorithm can detect *any* arbitrary stable predicate [3], which was an unsolved problem under the assumptions P1-P4. While many specialized algorithms are tailored to specific stable properties, such as deadlock, termination, and garbage, the following algorithms detect general stable predicates.

- The Marzullo-Sabel algorithm [13] can detect only *locally stable* predicates.
- The Schiper-Sandoz algorithm [15] can detect only *strong stable* predicates.
- Kshemkalyani-Singhal's two-phase algorithm [9], based on Ho-Ramamoorthy's algorithm [6], showed how to correctly detect deadlocks. A general method to detect stable properties was then outlined [9]. In essence, if a property does not change between the two serial phases of uncoordinated observations, the property must hold at some instant between the two phases. If it is stable, it must hold henceforth. While locally stable predicates can be detected satisfying assumptions (P1)-(P4) and without assuming FIFO channels, details of detecting arbitrary stable predicates are not given.

Neither Marzullo and Sabel [13] nor Schiper and Sandoz [15] showed any relationship between the classes of strong stable and locally stable properties. These existing algorithms can only detect some subclass of stable predicates, and do not satisfy (P1)-(P4). The proposed algorithm can detect *any* stable predicate.

Summary of Main Contributions:

1. The snapshot algorithms we propose for FIFO channels and for non-FIFO channels are linear in the number of messages, and satisfy (P1)-(P4).
2. The non-FIFO version of our snapshot algorithm can be used to detect locally stable predicates, under assumptions (P1)-(P4).
3. The FIFO version of our snapshot algorithm can be used to detect *any* stable predicate, under assumptions (P1) to (P4).

System Model:
An asynchronous execution in a distributed system is modeled as $(E, \prec)$, where $\prec$ is the causality relation [11] on the set of events E in the execution. $E = \bigcup_{i \in N} E_i$, where E_i is the totally ordered chain of event at process P_i. An event executed by P_i is denoted e_i. A *cut* C is a subset of E such that the events of a cut are downward-closed within each E_i. A *consistent cut* is a downward-closed subset of E. The system state after the events in a cut is a global state; if the cut is consistent, the corresponding system state is termed a *consistent global state*. An *execution slice* is defined as the difference of two cuts $D \setminus C$, where $C \subseteq D$. The slice is also referred to as a *suffix* of C with respect to D. When D is not explicitly specified, its default value is the execution E.

The execution history at a process P_i is a sequence of alternating states and local events, $\langle s_i^0, e_i^1, s_i^1, e_i^2, s_i^2, e_i^3, s_i^3, \ldots \rangle$. Events and messages of the snapshot algorithm form a superimposed control execution. Among the application events and messages, those that are relevant to the predicate of interest are the *relevant* events and messages, respectively. We assume that all events, variables, and messages recorded or logged are the relevant ones.

2 The Three-phase Uncoordinated Snapshot Algorithm

The proposed algorithm is inspired by the two-phase deadlock detection algorithm [9]. The main idea of our algorithm is as follows. The algorithm takes three serial uncoordinated 'snapshots' that may be inconsistent. A consistent global state that lies between the first and the second inconsistent 'snapshots' is computed with the help of the third 'snapshot' and some local processing.

Any process can initiate the algorithm which consists of three phases that are serially executed. The algorithm involves some local processing by the initiator. Each phase involves the initiator sending a request to each other process, and then the processes replying to the initiator. The initiator can communicate directly to/from the various processes, or a *wave algorithm* [5] can be used in conjunction with a superimposed topology such as a ring or a tree. The snapshot algorithm is independent of this detail.

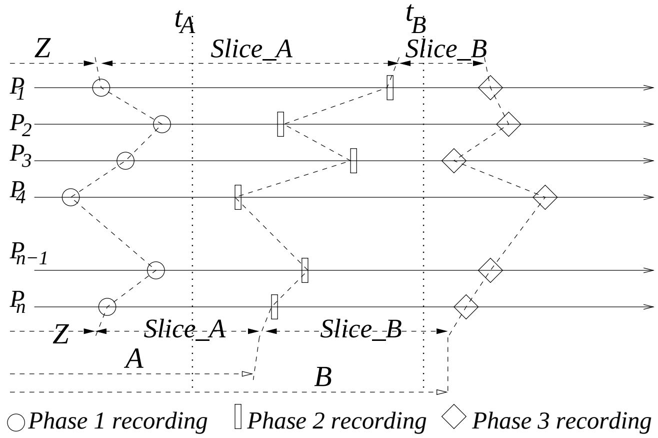

Fig. 1. Three-phase uncoordinated recording of a global snapshot. The initiator could be any of the processes.

Phase I: The initiator requests the processes to begin recording an execution slice. The global state from which processes begin recording is denoted Z, and is represented by the array $Z[1 \ldots n]$ at the initiator. The local states recorded in Z are not coordinated and may be inconsistent.

Phase II: The initiator then collects the slice of each process execution since the time each process began recording its slice and reported its local state in Phase I, until the time each process chooses to reply to the Phase II request. The local slice of each process that is reported to the initiator is stored by the initiator in array $Slice\_A[1 \ldots n]$. Each process begins to record the next slice, denoted *Slice_B*, after replying to the Phase II request.

Phase III: The initiator then collects the slice of each process execution since the time each process reported its local state in Phase II, until the time each process chooses to reply to the Phase III request. The local slice of each process that is reported to the initiator is stored by the initiator in array $Slice\_B[1 \ldots n]$. Based on *Slice_A* and *Slice_B*, the initiator computes a consistent global state.

Slice_B is used in two different ways, depending on whether channel states are to be recorded.

- If channel states are not to be recorded and the channels are non-FIFO, *Slice_B* is used to identify a consistent state within *Slice_A* by helping to eliminate states in *Slice_A* that are inconsistent.
- If channel states are also to be recorded and FIFO channels are assumed, then *Slice_B* is also useful to capture the channel states. In this case, the recording within *Slice_B* completes at each process when the messages sent by other processes to that process in (and before) *Slice_A* have been received. This condition is detectable using the control information

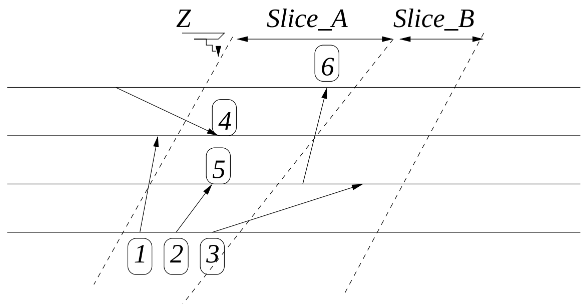

Fig. 2. Six types of events in *Slice_A*.

distributively sent to the initiator in the Phase II reply messages and then conveyed on the Phase III request received from the initiator.

The possibly inconsistent states collected by the initiator in Z, *Slice_A*, and *Slice_B* are illustrated in Figure 1. The initiator computes a consistent global state S, such that $Z \subseteq S \subseteq A$ using *Slice_A* and *Slice_B*. Specifically, observe that Z may be inconsistent because messages sent in *Slice_A* may have been received in Z. Also observe that due to the existence of global time instant t_A which is any time instant between the last recording of Phase I and the first recording of Phase II, no message sent in *Slice_B* could have been received before t_A. There exists at least one consistent cut in *Slice_A*, namely the cut at physical time t_A. However, as the application execution including its messages cannot be modified, and as timestamps are also not used in the algorithm, computing a consistent state S such that $Z \subseteq S \subseteq A$ is tricky. In *Slice_A*, there are six types of events (see Figure 2):

1. send event, for a message that gets delivered in Z
2. send event, for a message that gets delivered in *Slice_A*
3. send event, for a message that gets delivered after *Slice_A*
4. receive event, for a message that was sent in Z
5. receive event, for a message that was sent in *Slice_A*
6. receive event, for a message that was sent after *Slice_A*

To make Z consistent, we need to add that prefix from *Slice_A* that contains (i) all events of type (1) and no events of type (6), and (ii) the local states of processes are mutually consistent. Alternately, as A is also not consistent, A can be made consistent by subtracting that suffix that contains (i) all events of type (6), and (ii) further events to ensure that the resulting local states of processes are mutually consistent. With either approach, a consistent execution prefix exists, namely, the global state at t_A which satisfies both sets of conditions. Observe that all events of type (1) precede all events of type (6). Let $S(t)$ be the prefix of the execution at global time t. We now have the following.

- From Figure 1, we have $S(t_{Z\_end}) \subseteq S(t_A) \subseteq S(t_{A\_start})$, where t_{A_start} denotes the time instant of the first local recording of phase II among all the processes, and t_{Z_end} denotes the time instant of the last local recording of phase I among all the processes.
- Let S_{max} be Z + the largest prefix from *Slice_A* that does not include an event of type (6). Note that S_{max} may not be consistent.
- A tight lower bound on S_{max} is the value of $S(t_{A\_start})$. A tight upper bound on S_{max} is the value of $S(t_{A\_end})$, where t_{A_end} denotes the time instant of the last local recording of phase II among all the processes. Thus, $S(t_{A\_start}) \subseteq S_{max} \subseteq A \subseteq S(t_{A\_end})$.

The algorithm computes the largest consistent snapshot S such that $S(t_{A\_start}) \subseteq S \subseteq S_{max} \subseteq A \subseteq S(t_{A\_end})$, by removing the minimum slice suffix from S_{max} to get a consistent global state.

The third phase of recordings plays two roles.

- For both versions of the algorithm, *Slice_B* helps to identify S_{max} by identifying messages sent in *Slice_B* that were received in *Slice_A*.
- *Slice_B* also helps to identify the in-transit messages by ensuring that all the messages that have been sent up to and including in *Slice_A* are received before the recording of the end of *Slice_B*. This mechanism works only if FIFO channels are assumed.

The two versions of the algorithm are presented together in Figs. 3 and 4.

2.1 Consistent State Recording under non-FIFO channels

This section describes a global snapshot algorithm that works with non-FIFO channels and satisfies properties (P1) to (P4). This algorithm records a consistent global state but does not capture channel states.

Figure 3 gives the code for the three-phase processing. The underlined pseudocode and data structures are ignored by this (version of the) algorithm. Step (1) describes the processing followed by the initiator. Step (2) describes the processing at all the nodes. In the first phase, an acknowledgement is sufficient from the nodes to the initiator; the local states are not required to be reported. When *Slice_A* and *Slice_B* are recorded, observe the following.

- Only the relevant local and send/receive events, of interest to the application or predicate being monitored, are recorded in the log of the slice.
- Messages are not modified with sequence numbers to conform to requirements (P1)-(P4). In addition, no counters for sequence numbers for the messages sent or received, or for the event count, are required at processes. A hash or checksum ($O(1)$ space) computed on each message sent or received is stored in the log of the slice, to enable matching a message in the sender's log with the same message in the receiver's log. No messages are stored.
- Within the log of a slice at a process, sequence numbers are assigned to events. However, these sequence numbers are of significance only to that

process and within the slice. The sequence numbers have no global significance to the processes, or even within a process outside its slice. These numbers are used by the initiator to perform a simple ordering among the process events included in the slice.

After *Slice_A* and *Slice_B* have been collected at the end of the three phases, the initiator invokes procedure *Compute_Consistent_Snapshot* in Figure 4 to compute a consistent cut S from *Slice_A* and *Slice_B*. This cut S satisfies $S(t_{A\_start}) \subseteq S \subseteq S_{max} \subseteq A \subseteq S(t_{A\_end})$ and is computed by iteratively removing the minimum slice suffix from S_{max} to get a consistent global state. For convenience, this procedure represents each of the two slices as an array $[1 \ldots n]$ of an array of integers. For example, *Slice_A_events*$[1 \ldots i \ldots n][1 \ldots a_i]$, where *Slice_A_events*$[i][j]$ denotes the jth event at process P_i, represents *Slice_A*.

In Figure 4, lines (3a)-(3d) identify S_{max}. Line (3e) initializes the integer vector variables S, T, and U to S_{max}. These vectors denote global states but the integer values denoting sequence numbers of the states in the slice have significance local to the initiator only. For example, $S[i]$ was assigned by P_i relative to the start of the slice and represents the sequence number of a local state of P_i in the slice. Lines (3f-3g) initialize the vector variable V to the state at the end of *Slice_A*, namely, the state A. S is always set to the current known upper bound of the consistent global state that is sought. Vector V denotes the state that is the best known upper bound on the global state such that messages sent in the slice $V \setminus S$ may cause S to be inconsistent. Vectors T and U are working variables used to update S and V.

The main loop (3h)-(3q) updates S and V iteratively. In lines (3i)-(3l), U is used to track the prefix of the current S such that there are no inconsistencies caused by messages sent in $V \setminus S$. A message at the sender is matched with the same message at the receiver by comparing their hashes or checksums (line (3k)). If the message sent at *Slice_A_events*$[i][y]$ is received at *Slice_A_events*$[j][z]$, then $U[j]$ is updated to the minimum of its current value and $z - 1$ (line (3l)). An inconsistency, if any, is thus eliminated by removing the minimum suffix from the execution slice for the receiver P_j. However, messages sent in the slice $S \setminus U$ may still cause U to be inconsistent; this needs to be tested in the next iteration. Lines (3n)-(3p) initialize the values of S, T, U, and V for the next iteration. The procedure finishes when the loop (3i)-(3l) does not find any inconsistencies in the current value of S in line (3m).

2.2 Consistent State and Channel Recording under FIFO Channels

This section presents an enhanced algorithm that also records channel states if channels are FIFO. Figure 3 gives the code for the three-phase processing. Underlined pseudo-code and data structures are also executed by this version of the algorithm. Procedure *Compute_Consistent_Snapshot* in Figure 4 computes a consistent cut from *Slice_A* and *Slice_B*, and is common to both versions of the algorithm.

(variables at an initiator)
array of states: $Z[1 \ldots n]$; // Phase 1 recordings
array of sequence of events: $Slice\_A[1 \ldots n]$; // Phase 2 recordings
array of sequence of events: $Slice\_B[1 \ldots n]$; // Phase 3 recordings
array of int: $Global\_Sent[1 \ldots n, 1 \ldots n]$; $//Global\_Sent[i,j] \equiv$ # msgs, P_i to P_j
array of int: $Global\_Received[1 \ldots n, 1 \ldots n]$;
$//Global\_Received[i,j] \equiv$ # messages received by P_i from P_j
(variables at each process)
array of local events: *Slice_Log*; // log of local events
array of int: $Sent[1 \ldots n]$; // $Sent[k] \equiv$ # messages sent to P_k
array of int: $Received[1 \ldots n]$; // $Received[k] \equiv$ # messages received from P_k
array of int: $Must\_Receive[1 \ldots n]$;
// $Must\_Receive[k] \equiv$ # messages to be recd. from P_k before Phase III report

(1) Process P_{init} initiates the algorithm, where $1 \leq init \leq n$.
(1a) **send** *Request(Phase_1_Report)* to all P_j;
(1b) **await** *Report(Phase_1_Report)* from all processes;
(1c) $(\forall j)\ Z[j] \longleftarrow Phase\_1\_Report.State$ received from P_j;
(1d) $(\forall j)\ Global\_Received[j][1 \ldots n] \longleftarrow Phase\_1\_Report.Received[1 \ldots n]$;
(1e) **send** *Request(Phase_2_Report)* to all P_j;
(1f) **await** *Report(Phase_2_Report)* from all processes;
(1g) $(\forall j)\ Slice\_A[j] \longleftarrow Phase\_2\_Report.Slice\_Log$ received from P_j;
(1h) $(\forall j)\ Global\_Sent[j][1 \ldots n] \longleftarrow Phase\_2\_Report.Sent[1 \ldots n]$ from P_j;
(1i) **send** to all P_j, *Request(Phase_3_Report)* + $Global\_Sent[1 \ldots n][j]$ piggybacked;
(1j) **await** *Report(Phase_3_Report)* from all processes;
(1k) $(\forall j)\ Slice\_B[j] \longleftarrow$ Phase 3 report received from P_j;
(1l) $S \longleftarrow Compute\_Consistent\_Snapshot(Slice\_A, Slice\_B)$;
(1m) *Compute_In-transit_Messages*(S).

(2) Process P_j executes the following, where $1 \leq j \leq n$.
(2a) On receiving *Request(Phase_1_Report)* from P_{init},
(2b) **send** local state and $Received[1 \ldots n]$ in *Report(Phase_1_Report)* to P_{init};
(2c) Begin recording log of events in *Slice_Log*;
(2d) On receiving *Request(Phase_2_Report)* from P_{init},
(2e) **send** *Slice_Log* and $Sent[1 \ldots n]$ in *Report(Phase_2_Report)* to P_{init};
(2f) Reset *Slice_Log*;
(2g) On receiving *Request(Phase_3_Report)* and $Must\_Receive[1 \ldots n]$ from P_{init},
(2h) Await until, $(\forall k), Received[k] \geq Must\_Receive[k]$;
(2i) **send** *Slice_Log* in *Report(Phase_3_Report)* to P_{init};
(2j) Stop recording events in *Slice_Log*.

Fig. 3. Three-phase algorithm to record a global snapshot. Underlined code is executed if channel states are needed in a FIFO system.

Procedure *Compute_In_Transit_Messages* is used to compute the channel states. This procedure requires the data structures $Z[1 \ldots n]$ and integer arrays $Global\_Sent[1 \ldots n, 1 \ldots n]$ and $Global\_Received[1 \ldots n, 1 \ldots n]$ at the initiator during the processing of the algorithm. Integer vectors $Received[1 \ldots n]$,

(3) Process P_{init} executes *Compute_Consistent_Snapshot(Slice_A, Slice_B)***.**

array of int: S_{max}, S, T, U, V;

array of array of int: *Slice_A_events*$[1 \ldots i \ldots n][1 \ldots a_i]$;

array of array of int: *Slice_B_events*$[1 \ldots i \ldots n][1 \ldots b_i]$;

// Alternate representation of slices

for $i = 1$ **to** n **do**
    if *Slice_A*$[i][x+1]$ is the 1^{st} receive in *Slice_A[i]* of a msg. sent in *Slice_B* **then**
        $S_{max}[i] \longleftarrow x$
    else $S_{max}[i] \longleftarrow a_i$;
$S, T, U \longleftarrow S_{max}$;
for $i = 1$ **to** n **do**
    $V[i] \longleftarrow a_i$;
repeat
    for $i = 1$ **to** n **do**
        for $y = T[i] + 1$ **to** $V[i]$ **do**
            if message($Slice\_A\_events[i][y], Slice\_A\_events[j][z]$) **then**
                $U[j] \longleftarrow \min(U[j], z-1)$; // modify U to make it consistent
    if $T = U (= S)$ **then return**(S);
    $S \longleftarrow \min(T, U)$; // $S \equiv$ current upper bound on consistent state
    $V \longleftarrow \max(T, U)$; //current upper bound on source of inconsistency
    $T, U \longleftarrow S$;
forever.

(4) Process P_{init} executes *Compute_In-transit_Messages(S)***.**

$(\forall i \forall j)$ $transit(S[i], S[j]) \longleftarrow \emptyset$;
$(\forall i)$ compute *Global_Sent*$[i, 1 \ldots n]$ for $S[i]$ using *Slice_A*, *Global_Sent*$[i, 1 \ldots n]$;
for $j = 1$ **to** n **do**
    for each successive event e_j^x in *Slice_A*$[j][1 \ldots a_j]$ and *Slice_B*$[j][1 \ldots b_j]$ **do**
        if a message M was received from i (at this event with seq. # x) **then**
            Global_Received$[j, i] + +$;
            if *Global_Sent*$[i, j] \geq$ *Global_Received*$[j, i]$ **and** $x > S[j]$ **then**
                $transit(S[i], S[j]) \longleftarrow transit(S[i], S[j]) \cup \{M\}$.

Fig. 4. Finding a consistent state iteratively, and computing in-transit messages. Underlined code is executed if channel states are needed in a FIFO system.

Sent$[1 \ldots n]$, and *Must_Receive*$[1 \ldots n]$ must also be maintained at each node. *Sent*$[j]$ and *Received*$[j]$ track the count of the number of messages sent to and received from process P_j, respectively. The main idea is simple. The state of channel $\langle P_i, P_j \rangle$ in a global state containing local states $S[i]$ and $S[j]$ at the processes, denoted as $transit(S[i], S[j])$, is simply those messages sent by P_i till state $S[i]$ that are not received until state $S[j]$ at P_j. One important difference from the previous version is that sequence numbers used to count events at a process are not local to a slice, but local to the entire execution of that process. This is to capture in-transit messages for channel states. Such messages could have been sent before *Slice_A* and need to be detected. To compute the channel

Table 2. Complexity of the proposed non-inhibitory nonintrusive snapshot algorithm. Both *Slice_A* and *Slice_B* are *thin* slices, and their expected size is the same.

Metric	Recording snapshot (non--FIFO, no channel states)	Recording snapshot + channel states (w/ FIFO channels)
# messages	$6(n-1)$	$6(n-1)$
Msg. space (total)	$O(\|Slice\_A\|)$	$O(\|Slice\_A\|) + O(n^2)$
Time complexity (initiator)	$O(\|Slice\_A\|)+$ $O(1/n \times \|Slice\_A\|^2)$ $+O(n^2)$	$O(\|Slice\_A\|)+$ $O(1/n \times \|Slice\_A\|^2)$ $+O(n^2)$
Time complexity (non-initiator)	$O(\|Slice\_A\|)$	$O(\|Slice\_A\|) + O(n)$
Space complexity (initiator)	$O(\|Slice\_A\|)$	$O(\|Slice\_A\|) + O(n^2)$
Space complexity (noninitiator)	$O(1/n \cdot \|Slice\_A\|)$ avg.	$O(1/n \cdot \|Slice\_A\|)$ avg. $+O(n^2)$
Properties	No inhibition App. messages unmodified execution unmodified no log of history	No inhibition App. messages unmodified execution unmodified no log of history + *Sent*, *Received* vecs/process + *Global_Sent*, *Global_Receive* at init

state while satisfying conditions (P1) – (P4) and specifically that no sequence numbers can be tagged on messages, three issues need to be addressed.

1. Messages sent by P_i to P_j before state $S[i]$ must have reached P_j by the end of *Slice_B*.
   This is ensured by using the local *Sent* vector at each process and the *Global_Sent* array at the initiator. In the Phase II recording reported to the initiator, the *Sent* vectors reported (line (2e)) are used to populate *Global_Sent* (line (1h)). The Phase III request sent to each process contains the piggybacked information about how many messages have been sent to that process by other processes (line (1i)). A process postpones its Phase III recording of the end of *Slice_B* until all these number of messages, remembered in array $Must\_Receive$, have been delivered locally (line (2h)).
2. The set of messages sent by P_i to P_j up to the snapshot state $S[i]$, denoted here as $\mathcal{X}$, should be identifiable. There are two parts to this.
   – This set contains all the messages received by P_j from P_i with sequence numbers less than the value of $Sent[j]$ at $S[i]$. This value of $Sent[j]$ at $S[i]$ is computed (line (4b)) using *Global_Sent*, constructed from the Phase II report, and working backwards using the log *Slice_A*, also reported in Phase II. The resulting message count (i.e., the value of $Sent[j]$ at $S[i]$) is stored in-situ in the data structure $Global\_Sent[i, j]$ as it is updated.
   – The messages received by P_j are enumerated as per the sequence numbers assigned by P_i, in lines (4d)-(4e). The enumeration of the sequence numbers is done in lines (4c)-(4f) using *Global_Received*, reported in

Table 3. Comparing algorithms to detect stable predicates.

	Marzullo & Sabel [13]	Schiper & & Sandoz [15]	Proposed V.1	Proposed V.2
Detectable predicates	locally stable	strong stable	locally stable	all stable
overhead at nodes (= control msg overhead)	vector clock, $O(n)$ + entire log of msgs & events w/timestamps	vector clock, $O(n)$ + entire log of msgs & events w/timestamps	– event log in slice during 3-phase	*Sent/Received* $(O(n))$ + event & msgs. log in slice during 3-phase
App. msg overhead	vector clock, $O(n)$	vector clock, $O(n)$	–	–
processing	by initiator	by initiator	by initiator	by initiator
Channels	FIFO	non-FIFO	non-FIFO	FIFO
# control messages	$(n-1)$	$(n-1)$	$6(n-1)$	$6(n-1)$

Phase I to the initiator (line (1d)), and working forwards using the log *Slice_A* reported in Phase II and the log *Slice_B* reported in Phase III. The enumeration is done in-situ in $Global\_Received[j,i]$.

If $Global\_Received[j,i] \leq Global\_Sent[i,j]$ for a message, then that message belongs to $\mathcal{X}$.

3. The set of messages received by P_j from P_i after the snapshot state $S[j]$, denoted here as $\mathcal{Y}$, should be identifiable.
   These are the messages received from P_i at P_j in states numbered x such that $x > S[j]$.

From (2) and (3) above, $transit(S[i], S[j]) = \mathcal{X} \cap \mathcal{Y}$, is expressible as $\{M$ *recd by* P_j *at event* $x \mid Global\_Received[j,i] \leq Global\_Sent[i,j] \bigwedge x > S[j]\}$. Unlike the algorithm in Section 2.1, *Slice_Log*, *Slice_A*, *Slice_B* record messages for send and receive events if the contents of in-transit messages are required. The pseudo-code data structures do not reflect this for simplicity.

3 Complexity

The complexity analysis assumes a flat tree topology with the initiator as the root. A similar analysis can be conducted for the ring and more general tree topologies. Table 2 summarizes the complexity results. Note that the slices *Slice_A* and *Slice_B* are both *thin* slices, and their expected size is the same. Hence, the complexity is expressed in terms of *Slice_A* only. The expected width of *Slice_A* is the execution log that occurs in $\hat{rtt}_{max}$, the expected round-trip time between the two furthest nodes in the network. Let $max(owt_A)$ denote the maximum time for a message sent in *Slice_A* to reach its destination. The expected width of *Slice_B* is $max(rtt_{max}, owt_A)$ which is also $\hat{rtt}_{max}$.

4 Detecting Stable Predicates

The proposed algorithm to record a consistent global state can be used to detect any stable predicate. (See [7] for details.) Each process records the (possibly changing) values of the variables over which the predicate is defined, in *Slice_A*. When the initiator computes the consistent state S, it can also evaluate the predicate over these variables in state S. If the predicate is evaluated as true, then it is true and remains true henceforth because it is stable.

- Version 1 can detect locally stable predicates predicates.
- Version 2 can detect any stable predicate.

Table 3 compares the features and the complexities of the proposed algorithm with those of the algorithms by Marzullo-Sabel [13] and Schiper-Sandoz [15].

The full version of the results of this paper, including the correctness proofs and the complexity analysis, is in [7].

References

1. A. Acharya, B. R. Badrinath: Recording Distributed Snapshots Based on Causal Order of Message Delivery. Inf. Process. Lett. 44(6): 317-321 (1992)
2. S. Alagar, S. Venkatesan: An Optimal Algorithm for Distributed Snapshots with Causal Message Ordering. Inf. Process. Lett. 50(6): 311-316 (1994)
3. K. M. Chandy, L. Lamport: Distributed Snapshots: Determining Global States of Distributed Systems. ACM Trans. Comput. Syst. 3(1): 63-75 (1985)
4. C. Critchlow, K. Taylor: The Inhibition Spectrum and the Achievement of Causal Consistency. Distributed Computing 10(1): 11-27 (1996)
5. J.-M. Helary: Observing Global States of Asynchronous Distributed Applications. WDAG 1989: 124-135
6. G. S. Ho, C. V. Ramamoorthy: Protocols for Deadlock Detection in Distributed Database Systems. IEEE Trans. Software Eng. 8(6): 554-557 (1982)
7. A. Kshemkalyani, B. Wu: Detecting Arbitrary Stable Properties Using Efficient Nonintrusive Snapshots. Univ. of Illinois at Chicago, Tech. Report UIC-CS-03-05 (2005)
8. A. Kshemkalyani, M. Raynal, M. Singhal: An Introduction to Snapshot Algorithms in Distributed Computing. Distributed Systems Engineering 2(4): 224-233 (1995)
9. A. Kshemkalyani, M. Singhal: Correct Two-Phase and One-Phase Deadlock Detection Algorithms for Distributed Systems. IEEE SPDP 1990: 126-129
10. T.-H. Lai, T. Yang: On Distributed Snapshots. Inf. Process. Lett. 25(3): 153-158 (1987)
11. L. Lamport: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM 21(7): 558-565 (1978)
12. H. F. Li, T. Radhakrishnan, K. Venkatesh: Global State Detection in Non-FIFO Networks. IEEE ICDCS 1987: 364-370
13. K. Marzullo, L. S. Sabel: Efficient Detection of a Class of Stable Properties. Distributed Computing 8(2): 81-91 (1994)
14. F. Mattern: Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. J. Parallel Distrib. Comput. 18(4): 423-434 (1993)
15. A. Schiper, A. Sandoz: Strong Stable Properties in Distributed Systems. Distributed Computing 8(2): 93-103 (1994)
16. M. Spezialetti, P. Kearns: Efficient Distributed Snapshots. IEEE ICDCS 1986: 382-388
17. S. Venkatesan: Message-Optimal Incremental Snapshots. IEEE ICDCS 1989: 53-60