Detecting Arbitrary Stable Properties Using Efficient Snapshots

Ajay Kshemkalyani, *Senior Member*, *IEEE*, and Bin Wu

Abstract—A stable property continues to hold in an execution once it becomes true. Detecting arbitrary stable properties efficiently in distributed executions is still an open problem. The known algorithms for detecting arbitrary stable properties and snapshot algorithms used to detect such stable properties suffer from drawbacks such as the following: They incur the overhead of a large number of messages per global snapshot, or alter application message headers, or use inhibition, or use the execution history, or assume a strong property such as causal delivery of messages in the system. We solve the problem of detecting an arbitrary stable property efficiently under the following assumptions: P1) The application messages should not be modified, not even by timestamps or message coloring. P2) No *inhibition* is allowed. P3) The algorithm should not use the message history. P4) Any process can initiate the algorithm. This paper proposes a family of nonintrusive algorithms requiring $6(n-1)$ control messages, where n is the number of processes. A three-phase strategy of uncoordinated observation of local states is used to give a consistent snapshot from which any stable property can be detected. A key feature of our algorithms is that they do not rely on the processes continually and pessimistically reporting their activity. Only the relevant activity that occurs in the *thin* slice during the algorithm execution needs to be examined.

Index Terms—Distributed system, global state, stable predicate, stable property, distributed snapshot.

1 Introduction and Problem Definition

1.1 Problem Formulation

A snapshot of a distributed system represents a consistent global state of the system [4]. Recording distributed snapshots of an execution is a fundamental problem in asynchronous distributed systems and is used for observing various properties of interest [1], [2], [4], [7], [11], [14], [16], [19], [22], [23], [25].

A restricted class of properties, termed stable properties, are monotonic in the following sense: A *stable property* continues to hold in an execution once it becomes true [4]. Commonly occurring examples of *stable properties* are deadlock, termination, and system garbage. Many specialized algorithms are tailored to specific stable properties, such as to detect deadlock or termination or to perform garbage collection; see [17], [21] for a survey of such algorithms. However, our objective is to devise an efficient algorithm to detect an arbitrary stable property. While global snapshot algorithms can always be used to detect stable properties, they suffer from some or all of the following drawbacks: They are expensive and generally require a large number of messages per snapshot [4], [23], [22], [25], [7], [14], [16], or they use freezing of the application execution [7], or they alter application message headers [14], [16], [19], or they assume some strong properties such as causal message delivery in the system [1], [2].

• The authors are with the Department of Computer Science, University of Illinois at Chicago, 851 South Morgan Street, Chicago, IL 60607. E-mail: ajayk@cs.uic.edu, bwu3@uic.edu.

Manuscript received 30 Apr. 2005; revised 10 Feb. 2006; accepted 22 Jan. 2007; published online 13 Feb. 2007.
Recommended for acceptance by R. Schlichting.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0116-0405.
Digital Object Identifier no. 10.1109/TSE.2007.1000.

Consider a distributed system that 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 $l = |L|$. We solve the open problem of efficiently detecting an arbitrary stable property under the following assumptions (P1-P4):

P1. The application messages should not be modified, not even by timestamps [6], [18] or message coloring.

P2. No freezing or inhibition [5] in the application execution is allowed. This rules out all three forms of inhibition: inhibition of send events, inhibition of receive events, and inhibition of local noncommunication events, i.e., internal events.

P3. The algorithm should not use the log of the message history.

P4. Any process can initiate the algorithm.

The known algorithms for detecting stable properties can detect only a subclass of stable properties—such as *locally stable* properties [17], *strongly stable* properties [14], and *strong stable* properties [21]—and may not satisfy all of (P1-P4).

1.2 Prior Work

A property is an abstract characteristic of an execution. An explicit formulation of a property in terms of process states and variables is a *predicate*. For example, deadlock is a property. A deadlock predicate is its explicit formulation, in terms of the *blocked status* variable, *outgoing wait-for edge* variables, and *incoming wait-for edge* variables at each process. Keeping this association in mind, we will henceforth use the term "predicates" rather than "properties." There are only the following algorithms that try to detect *general* stable predicates that are not tailored for a specific predicate:

- The algorithm by Marzullo and Sabel [17] can detect only *locally stable* predicates. A *locally stable predicate* is a stable predicate such that, once it becomes true, none of the variables used in the predicate formulation changes in value. Stated differently, the predicate can be determined only from the states of processes and no process involved in the predicate can change its state once the predicate holds.

The algorithm by Marzullo and Sabel requires the use of a (modified) vector clock [6], [18] that modifies application messages. The entire log of the execution history (of events that can affect the predicate) is implicitly used by a pessimistic sending condition followed by the processes. Specifically, the predicate detection algorithm receives the local histories from each process since the last time they were reported. It then computes the latest *consistent subcut*, which is defined as the collection of the latest reported states of processes that are consistent with each other. The predicate is evaluated for this latest consistent subcut. The protocol is sound—if ϕ holds for the latest consistent subcut, ϕ holds thenceforth. To show completeness, let GS^f_ϕ be the first global state in which ϕ holds, and let the local state at P_i in this global state be $GS^f_\phi[i]$. As ϕ is locally stable, all events that happen after $GS^f_\phi[i]$ are not relevant—this is the key here—and P_i's state will not change further. As a result, no message sent after $GS^f_\phi[i]$ can cause inconsistency. Once $GS^f_\phi[i]$ is collected (for each P_i), the $GS^f_\phi[i]$ will appear on a latest consistent subcut and the locally stable predicate will be detected. The protocol implicitly assumes that, once the predicate ϕ becomes true, the involved processes will stop sending messages to report local activities, without actually knowing that ϕ has become true.

Termination, deadlock, and global virtual time can be formulated as locally stable predicates. The predicate "there are at most k tokens" is not locally stable, as shown in [17]. In an execution where variables $x_1, \ldots x_k$ at processes $P_1, \ldots P_k$, respectively, take nondecreasing positive values, the predicate $x_1 \times \ldots \times x_k > c$ can be seen to be not locally stable.

The protocol [17] is then used to derive even simpler protocols for deadlock and other locally stable properties, which are instantiations of the general protocol.

- The algorithm by Schiper and Sandoz [21] can detect only *strong stable* predicates. They define a *consistent projection of a cut* as a subcut containing those local process states that are mutually consistent with each other. A *strong stable predicate* is a stable predicate that can always be evaluated on the consistent projection of any global state, consistent or not. It is sufficient to evaluate the predicate on a consistent projection of any global state to determine if the predicate is true for the execution [21]. The algorithm implicitly assumes that, once the predicate becomes true (for a consistent projection of a cut over the "minimal" number of processes necessary), the involved processes will stop sending messages to report local activities without actually knowing that the predicate has become true.

  In the basic version of the Schiper-Sandoz algorithm, processes continually report their new states to a central process that keeps evaluating consistent projections of the latest constructed global state until the evaluation leads to *true*. Thus, all processes follow a pessimistic sending rule as in the Marzullo-Sabel algorithm. The algorithm requires the use of a (modified) vector clock that modifies application messages to construct the consistent projections. The entire log of the execution history (of events that can affect the value of the predicate) is also implicitly used to construct the consistent projections for all global states, in order to detect this class of stable predicates. Specifically, "each process furnishes an unbounded sequence of local contributions ... (until termination or predicate detection)" [21].

  The protocol is sound—if ϕ holds for the latest consistent projection, ϕ holds thenceforth. To show completeness, observe that once ϕ holds, by definition, the predicate will evaluate to *true* on the consistent projection of the first global state when ϕ holds.

  Termination and deadlock can be formulated as strong stable predicates. Distributed garbage is not strong stable, as shown in [21].
- Lai and Yang [14] defined the class of *strongly stable predicates*. A strongly stable predicate is a stable predicate such that, if the predicate is true on some cut, consistent or not, the predicate must remain true for all subsequent cuts.
- Based on the Ho-Ramamoorthy two-phase deadlock detection algorithm [10], Kshemkalyani-Singhal gave a two-phase technique [12] that showed how to correctly detect deadlocks and how to detect stable properties. A possibly inconsistent snapshot of the execution is taken; if the local predicates indicate a deadlock, a second possibly inconsistent snapshot is initiated. If the *same* deadlock also holds in this second snapshot, it is concluded that deadlock indeed exists because there is an instant of time between the two snapshots when the deadlock also held. A generalized method to detect stable properties was then outlined [12]. While all locally stable predicates can be detected satisfying our assumptions (P1-P4) and without assuming FIFO channels, it is not clear how arbitrary local predicates would be detected.
- Recently, Atreya et al. [3] proposed an algorithm to detect a locally stable predicate by repeatedly taking possibly inconsistent snapshots as long as 1) there is one consistent state between the two possibly inconsistent snapshots, and 2) the values of the variables do not change in the two snapshots. This is, in essence, the Ho-Ramamoorthy algorithm [10] as modified and corrected in [12]. The generalization to detect locally stable properties other than deadlock was more formally shown than in [12]. Only locally stable predicates can be detected.

TABLE 1
Comparing Algorithms to Detect Stable Predicates

	Marzullo & Sabel [17]	Schiper & Sandoz [21]	Proposed v.1	Proposed v.2
Detectable predicates	locally stable	strong stable	locally stable	all stable
Overhead at nodes (= control msg. overhead)	vector clock, $O(n)$ + log of msgs & events w/timestamps	vector clock, $O(n)$ + log of msgs & events w/timestamps	– events log in slices ***only during 3-phase***	*Sent* & *Received*, $O(n)$ + [events & msgs log in slices ***only during 3-phase***]
Persistent data structures	vector clocks	vector clocks or counters	–	counters
Entire history used	yes	yes	no	no
Application 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 msgs.	$(n-1)$	$(n-1)$	$6(n-1)$	$6(n-1)$

The messages and events in the logs are only the relevant events in all cases. The Sent *and* Received *arrays and the notion of slice are explained in Section 3.*

- Helary et al. [8] proposed an algorithm to detect a "fairly general" class of stable properties without characterizing this class any further. Informally, it uses local control variables at each process to track 1) whether a process is active or passive, 2) whether each incident channel is empty, and 3) whether each channel is *closed*, i.e., whether or not a message will ever be received on it. Whenever these variables change value so as to suggest the stable property might be true, the process sends the information (along with counts of the number of messages sent to and received from all processes) to the monitor process. The monitor process pieces together a global state using such information from all processes and if the stable predicate is true, it gets detected. The class of stable properties that can be detected is not characterized in [8], but they give examples of deadlock detection and termination detection. It appears that the algorithm can detect only those properties that can be expressed in terms of the quiescence of channels, i.e., variables that can track items 1, 2, and 3 above.

Neither Marzullo and Sabel [17] nor Schiper and Sandoz [21] showed any relationship between the classes of *strong stable* and *locally stable* predicates. Schiper and Sandoz conjectured that they are similar.

A strongly stable predicate differs from a strong stable predicate in that the predicate is evaluated over *any cut*, consistent or not, and variables at processes not in the consistent projection are also considered in evaluating the predicate. The predicate $\phi =_{def} x_1 + x_2 + \ldots + x_n \geq k$, where each x_i is a monotonically nondecreasing variable at process P_i, is strongly stable but is not strong stable. Deadlock is strong stable but not strongly stable. The above strongly stable predicate ϕ is not locally stable.

The above surveyed algorithms can only detect some subclass of stable predicates—for example, they cannot detect "there are at most k tokens" in a system where tokens can be lost and initially there are more than k tokens. Furthermore, some of these algorithms [14], [17], [21] modify all application messages with vector timestamps and use the log of the message history to report to the algorithm. Hence, they do not meet all the assumptions P1-P4 above.

1.3 Main Idea

The algorithm proposed in this paper can be used to detect *any* stable predicate and satisfies properties P1-P4 above. The proposed algorithm requires $6(n-1)$ control messages, where n is the number of processes in the system. A simple and elegant three-phase strategy of uncoordinated observation of local states is used to give a consistent snapshot from which any stable property can be detected. Minimal data structures are used *only during the execution of the three phases*; the executions between the first and second phases, and between the second and third phases, are termed as *slices*. Two versions of the algorithm are presented:

Version 1. The first version records consistent process states without requiring the channels to be FIFO.

Version 2. The second version records process states and channel states consistently but requires FIFO channels. (Two variants of this version of the algorithm are also presented in a later section.)

Table 1 compares the features and the complexities of the proposed algorithm with those of the algorithms in [17], [21].

Stable predicate detection is a special case of global state detection. Our stable predicate detection algorithms follow directly from the algorithms we propose for recording a distributed snapshot efficiently. Since the seminal elegant algorithm of Chandy and Lamport [4], which is a noninhibitory algorithm that requires FIFO channels and $2l$ messages for recording the snapshot, plus additional messages for assembling the snapshot, several other algorithms have been proposed. A good survey and comparison of such algorithms is given in [11]. In our approach, a simple and elegant three-phase strategy of uncoordinated observation of local states gives a consistent distributed snapshot. The first version of the snapshot algorithm (v.1) records consistent process states without requiring FIFO channels and without using any form of

TABLE 2
Features of Global Snapshot Algorithms

Algorithm	Channels	Inhibitory	Application messages modified	Number of control messages	Snapshot collection needed	Message history used
Chandy-Lamport [4] Sridhar-Sivilotti [23]	FIFO	N	N	$O(n^2)$	Y	N
Spezialetti-Kearns [22]	FIFO	N	N	$O(n^2)$	Y	N
Venkatesan [25]	FIFO	N	N	$O(n^2)$	Y	N
Helary [7] (wave sync.)	FIFO	Y	N	$O(n^2)$	Y	N
Helary [7] (wave sync.)	non-FIFO	Y	N	$O(n^2)$	Y	N
Lai-Yang [14]	non-FIFO	N	Y (p.b.)	$O(n^2)$	Y	Y
LRV [16]	non-FIFO	N	Y (p.b.)	$O(n^2)$	Y	Y
Mattern [19]	non-FIFO	N	Y (p.b.)	$O(n)$	Y	N
Acharya-Badrinath [1]	CO	N	N	$2n$	N	N
Alagar-Venkatesan [2]	CO	N	N	$3n$	N	N
Proposed snapshot v.1 (w/o channel states)	non-FIFO	N	N	$6n$	N	N
Proposed snapshot v.2	FIFO	N	N	$6n$	N	N

The acronym p.b. denotes that control information is piggybacked on the application messages.

message send/receive or event counters. The second version (v.2) 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 [5]. 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.

We compare the properties of the proposed snapshot algorithm (both versions) with the properties of the existing algorithms in Table 2. Existing algorithms can be classified into three broad classes:

- The baseline Chandy-Lamport algorithm [4], and its variants, the Spezialetti-Kearns algorithm [22], Venkatesan's algorithm [25], the Sridhar-Sivilotti algorithm [23], and Helary's wave synchronization algorithm [7] assume FIFO channels, are noninhibitory, do not modify messages, and require $O(n^2)$ control messages to record the snapshot. Additional messages are required to collect the locally recorded snapshot views.
- For non-FIFO channels, Helary's inhibitory algorithm [7], the Lai-Yang algorithm [14], the Li-Radhakrishnan-Venkatesh (LRV) algorithm [16], and Mattern's termination-detection-based algorithm [19] have been proposed. All these (except Helary [7], which is inhibitory) require application messages to be modified. Additional messages are required to collect the locally recorded snapshots. These algorithms (except Mattern's [19], which uses termination detection, and Helary's [7], which uses inhibition) require entire message histories to be recorded and piggybacked as part of the snapshot recording.
- Algorithms such as those by Acharya and Badrinath [1] and Alagar and Venkatesan [2] assume causally ordered (CO) message delivery and a single initiator and require $O(n)$ control messages. They do not require additional messages to explicitly collect the locally recorded snapshots because the snapshot gets assembled at the initiator as part of the algorithm. However, causally ordered channels are expensive. They require $O(n^2)$ overhead and delivery of messages may have to be delayed.

Other variants that require the use of vector timestamps [6], [18] are not considered here because of their high cost. As related work, Helary and Raynal gave a broad framework [9] based on iterative guarded wave sequences, which suggests a way to derive algorithms for specific predicates.

1.4 Summary of Main Contributions

1. As seen from Table 2, the snapshot algorithm we propose for FIFO channels is linear in the number of messages, (P1) does not modify the application messages, (P2) is noninhibitory, (P3) does not require the message history, and (P4) allows any process to initiate the algorithm. This is an important contribution to a fundamental problem.
2. The non-FIFO version of our snapshot algorithm can be used to detect locally stable predicates, under assumptions P1 through to P4.
3. The FIFO version of our snapshot algorithm can be used to detect any stable predicate, assuming P1 through P4. This algorithm is compared with other stable predicate detection algorithms in Table 1.

Outline. Section 2 gives the system model. Section 3 presents the proposed 3-phase snapshot algorithm (version 1) that works with non-FIFO channels but does not record channel states. Section 4 shows how to adapt this algorithm to record channel states if FIFO channels are assumed (version 2). Section 4 gives the complexity analysis. Section 5 shows how the respective algorithms can be used to detect any stable predicate (or a locally stable predicate) under the assumptions P1 through P4. Section 7 discusses additional variants of the proposed algorithms. Section 8 gives the conclusions.

2 System Model

The system model of the asynchronous reliable distributed system is as follows: We assume all messages are delivered

```
record event
        process : integer {1 ... n};
        operation : string { "send", "receive", or the internal operation };
        partner : integer {1 ... n, ⊥};
        hash : string; initialized to ⊥;
        next : pointer; initialized to ⊥;
        previous : pointer initialized to ⊥;
        seqno : integer;  // sequence number of event, used only when channel states are to be recorded
        M : string;           // used only when channel states are to be recorded
end.
```

Fig. 1. Data structure of the **event** record. The event sequence is a doubly linked list.

in finite time. An asynchronous execution in a distributed system with n processes is modeled as an execution $(E, \prec)$, where $\prec$ is the causality relation 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 events at process P_i. An event e executed by P_i is denoted e_i. The potential causality or the "happens before" relation $\prec$ on E is the transitive closure of the local ordering relation on each E_i and the ordering imposed by message send events and message receive events [15]. The execution history at a process P_i is a sequence

$$\langle s_i^0, e_i^1, s_i^1, e_i^2, s_i^2, e_i^3, s_i^3, \ldots \rangle$$

of alternating states and local events, where s_i^0 is the initial state of process P_i and event e_i^x causes a state transition from state s_i^{x-1} to state s_i^x. Due to the bijective mapping among the noninitial states and the events at a process, reference to an event implicitly identifies the corresponding state.

A *cut* C is a subset of E such that if $e_i \in C$, then

$$(\forall e_i')\ e_i' \prec e_i \Longrightarrow e_i' \in C.$$

In contrast, a consistent cut C is a subset of E such that if $e \in C$ then $(\forall e')\ e' \prec e \Longrightarrow e' \in C$. The system state after the events in a cut is a global state; if the cut is consistent, the corresponding system state is a *consistent global state* and denotes a meaningful observation of a global state [4]. Formally, a global snapshot or global state GS consists of

- a set of local states $\{s_1^{x_1}, s_2^{x_2} \ldots s_n^{x_n}\}$ and
- a set of the states of channels $C_{i,j}$ $(\forall i \forall j)$, where $state(C_{ij}) = transit(e_i^{x_i}, e_j^{x_j})$, the set of messages sent by P_i up to event $e_i^{x_i}$ and not received by P_j up to event $e_j^{x_j}$.

An *execution slice* is defined as the difference of two cuts $D \setminus C$, where $C \subseteq D$. This definition of a slice is somewhat different from the traditional definition in software engineering [26]. The slice is also referred to as a *suffix* of C with respect to D. The default value of C is $\emptyset$, in which case the slice is the cut D. Observe that each cut D is also a slice (i.e., $D \setminus \emptyset$). A slice D which is a set has a natural representation as an *array* $D[1 \ldots n]$ *of sequences of events*, one sequence per process. This representation captures the local order of events at each process without additional overhead. The reference to a slice—to its definition as a set or to its representation as an array of sequence of events—should be clear from context. The *suffix* and *prefix* of a slice $D[1 \ldots n]$ have a natural interpretation, namely, as the suffix or prefix of each $D[i]$, $1 \leq i \leq n$.

A *stable predicate* is a predicate that remains *true* once it becomes *true* [4]. Let $\models$ denote the satisfaction operator for a formula. Formally, if predicate ϕ is stable, then

$$C \models \phi \texttt{ implies that } (\forall D \mid D \supseteq C)\ D \models \phi.$$

Events and messages of the snapshot algorithm form a superimposed control execution that does not affect the application execution E or the partial order $(E, \prec)$ which is defined only on the application events. Among the application events and messages, the events and messages that are relevant to the predicate of interest are termed as *relevant* events and messages, respectively. We assume that all events, variables, and messages recorded in the algorithms are only the *relevant* ones. For a send or receive event that occurs at process indexed i, unless otherwise specified, we use $\tilde{i}$ to denote index of the process at which the corresponding receive or send event occurs.

3 Three-Phase Algorithm: Local States, Non-FIFO Channels

3.1 Strategy

The snapshot algorithm was inspired by the two-phase deadlock detection algorithm [10], [12]. The algorithm has three serially executed phases, in each of which uncoordinated "snapshots" that may be inconsistent are recorded. A consistent global state that lies between the first and the second inconsistent "snapshots" is computed with the help of the third "snapshot." Any process can initiate the algorithm, but it has to do some local processing. 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* [20] can be used in conjunction with a superimposed topology such as a ring or a tree. The snapshot algorithm is independent of this detail.

Phase I. The initiator requests the processes to begin recording an execution slice. Each process replies to the initiator when it receives the Phase I request, by piggybacking the local state on this reply. Each process then begins to record a slice as a sequence of (relevant) events, where each event has the data structure shown in Fig. 1. Specifically,

- For send and receive events, the hash of the message (such as SHA-1) and the corresponding receiver/sender are also recorded in fields *hash* and *partner*, respectively.

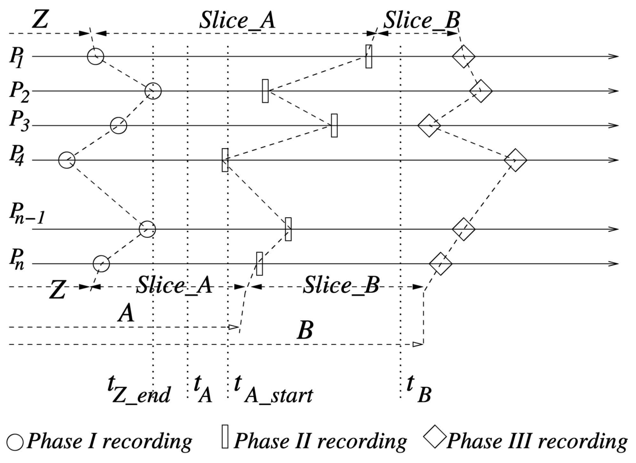

Fig. 2. Timing diagram to illustrate three-phase uncoordinated recording of a global snapshot. The initiator could be any of the processes.

- The field *seqno* gives the sequence number of the event at its process. The field M records the message received. These fields are used only if channel states are to be recorded (Version 2 of the algorithm).

The cut after which processes begin recording the slice is denoted Z. The state after this cut is represented by the array $State\_Z[1 \ldots n]$ at the initiator. The local states in $State\_Z$ may represent an inconsistent global state due to the uncoordinated nature of their recording.

Phase II. The initiator waits for the Phase I replies from all other processes. It then sends a Phase II request to all the processes. When a process receives a Phase II request, it reports the slice of its execution since the time it began recording its slice in Phase I until the time it 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 $Slice\_A[1 \ldots n]$. The cut containing all the events up to the local time at which each process replies to the Phase II request is denoted as A. After sending the Phase II reply, each process starts recording the next slice.

Phase III. The initiator waits to receive the Phase II reply from each process. It then sends a Phase III request to all the processes. When a process receives a Phase III request, it reports the slice of its execution that was recorded since the time it replied to the Phase II request until the time it 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 $Slice\_B[1 \ldots n]$. The cut containing all the events up to the local time at which each process replies to the Phase III request is denoted as B.

The possibly inconsistent values that are reported to the initiator in $State\_Z$, $Slice\_A$, and $Slice\_B$ are illustrated in Fig. 2. The initiator constructs a consistent global cut S, such that $Z \subseteq S \subseteq A$ using $Slice\_A$ and $Slice\_B$. In $Slice\_A$, there are six types of send and receive events (see Fig. 3).

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,

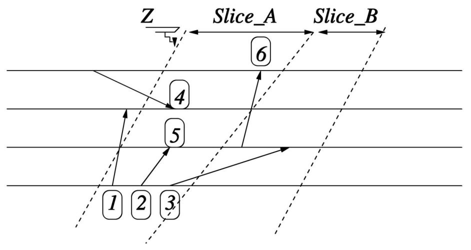

Fig. 3. Timing diagram to illustrate six types of events in $Slice\_A$.

5. receive event, for a message that was sent in $Slice\_A$, and
6. receive event, for a message that was sent after $Slice\_A$. Due to the existence of time instant t_B as shown in Fig. 2, the message must have been sent in $Slice_B$.

This suggests two approaches to construct a consistent cut S:

Extend Z: Extend Z to a consistent cut S by adding that prefix from $Slice\_A$ satisfying the following:

- The prefix contains all events of type 1 and no events of type 6, and
- $Z \cup \{\text{events in the prefix}\}$ is a consistent cut.

Reduce A: As cut A may not be a consistent cut, we can construct a consistent cut S based on A by subtracting the suffix that contains

- all events of type 6 and
- additional events to ensure that $A \setminus \{\text{events in suffix}\}$ is a consistent cut.

Observe that in real time, all events of type 1 precede all events of type 6. Define t_{A_start}, t_{Z_end}, and t_A (see Fig. 2) as follows:

- Let t_{A_start} denote the time instant of the first local recording of Phase II among all the processes.
- Let t_{Z_end} denote the time instant of the last local recording of Phase I among all the processes.
- Let t_A be any instant in global time such that $t_{Z\_end} < t_A < t_{A\_start}$.

Let $S(t)$ be the cut at global time t. The cut $S(t_A)$ which is a consistent cut satisfying the conditions on cut S above (for both the "Extend Z" and "Reduce A" approaches) must exist.

In the exposition of our algorithm (Version 2), we follow the "Reduce A" approach. In Section 7, we give two variants of Version 2 of the algorithm:

Version 2A. This variant leverages the use of the data structures more efficiently than Version 2.

Version 2B. This variant uses the "Extend Z" approach. We now define the cut S_{max}.

- $S_{max} = Z \bigcup \{$ events in the largest prefix from $Slice\_A$ that does not include an event of type 6 $\}$.

See Fig. 4. Note that S_{max} may not be consistent. The cut S_{max} includes $S(t_{A\_start})$ and A includes S_{max}.

The algorithm constructs the largest consistent cut S such that $S(t_{A\_start}) \subseteq S \subseteq S_{max} \subseteq A$, in two steps:

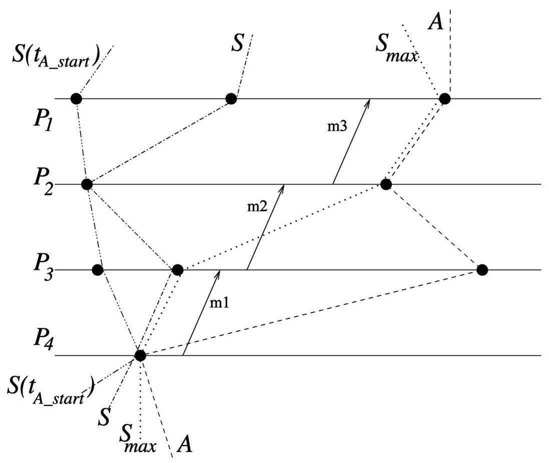

Fig. 4. Timing diagram to illustrate example relationships among cuts A, S_{max}, S, and $S(t_{A\_start})$. For each cut, only the latest event at each process is shown.

1. $Slice\_B$ helps to construct S_{max} by identifying messages sent in $Slice\_B$ that were received in $Slice\_A$. For example, in Fig. 4, the message $m1$ has its send event in $Slice\_B$. The minimum slice suffix from $Slice\_A[i]$ $(\forall i)$ that contains the receive event of each message sent in $Slice\_B$ is removed to construct S_{max}. In the example, S_{max} is therefore different from A. S_{max} may still be inconsistent due to receive events for messages that were sent from $Slice\_A$. In the example, the message $m2$ causes S_{max} to be inconsistent.
2. The algorithm then removes the minimum slice suffix from S_{max} to remove receive events that cause inconsistency. This is done iteratively to construct a consistent cut S. In the example, messages such as $m2$ and then $m3$ need to be removed.

The set of all cuts forms a lattice [18]. As S_{max} is defined to be the cut that is the largest subset of A and does not contain any receive events for which messages were sent in $Slice\_B$, S_{max} is unique due to the lattice property. The set of all consistent cuts also forms a lattice. As S is defined to be the largest consistent cut within S_{max}, S is also unique. As A, S_{max}, and S are all uniquely defined, the minimum slice suffixes to be removed to construct S_{max} and S are also well-defined.

3.2 Three-Phase Algorithm

Fig. 5 gives the code for the three-phase processing. Step 1 describes the processing at the initiator. Step 2 describes the processing at all the nodes. When $Slice\_A$ and $Slice\_B$ are recorded, note the following:

- Messages are not modified with sequence numbers to conform to P1. In addition, no counters for sequence numbers for the messages sent or received, or for the event count, are used at processes. A hash or checksum 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. By choosing an appropriate hash function, the probability of collisions of the hash values can be made arbitrarily small.

```
(variables at an initiator)
$State\_Z[1 \dots n]$: array of states;                    // Phase 1 state recording, stored at initiator
$State\_S[1 \dots n]$: array of states;                    // states in the constructed consistent cut
$Last\_S[1 \dots n]$: array of pointers to event;          // pointers to last events in constructed consistent cut
$Slice\_A[1 \dots n]$: array of sequence of event;         // Phase 2 slice recordings, stored at initiator
$Slice\_B[1 \dots n]$: array of sequence of event;         // Phase 3 slice recordings, stored at initiator
(variables at each process)
$Slice\_Log$: sequence of event;                            // log of local events within a slice

(1) Process $P_{init}$ initiates the algorithm, where $1 \le init \le n$.
(1a) send $Request\_Phase\_1\_Report$ to all $P_j$;
(1b) await $Phase\_1\_Report(state_j)$ from each process $P_j$;
(1c)     $(\forall j)$ $State\_Z[j] \longleftarrow state_j$ received from $P_j$;
(1d) send $Request\_Phase\_2\_Report$ to all $P_j$;
(1e) await $Phase\_2\_Report(Slice\_Log_j)$ from each process $P_j$;
(1f)     $(\forall j)$ $Slice\_A[j] \longleftarrow Slice\_Log_j$ received from $P_j$;
(1g) send $Request\_Phase\_3\_Report$ to all $P_j$;
(1h) await $Phase\_3\_Report(Slice\_Log_j)$ from each process $P_j$;
(1i)     $(\forall j)$ $Slice\_B[j] \longleftarrow Slice\_Log_j$ received from $P_j$;
(1j) $Compute\_Consistent\_Snapshot(Slice\_A, Slice\_B)$.          // snapshot computed in $State\_S$

(2) Process $P_j$ executes the following, for each $j$ such that $1 \le j \le n$.
(2a) On receiving $Request\_Phase\_1\_Report$ from $P_{init}$,
(2b)     send $Phase\_1\_Report(state_j)$ to $P_{init}$;
(2c)     Begin recording sequence of events in $Slice\_Log$;
(2d) On receiving $Request\_Phase\_2\_Report$ from $P_{init}$,
(2e)     send $Phase\_2\_Report(Slice\_Log)$ to $P_{init}$;
(2f)     Reset $Slice\_Log$; begin recording sequence of events in $Slice\_Log$;
(2g) On receiving $Request\_Phase\_3\_Report$ from $P_{init}$,
(2h)     send $Phase\_3\_Report(Slice\_Log)$ to $P_{init}$;
(2i)     Stop recording events in $Slice\_Log$.
```

Fig. 5. Three-phase algorithm to record a global snapshot. Continued in Fig. 6.

```
(3) Process $P_{init}$ executes Compute_Consistent_Snapshot(Slice_A, Slice_B).
$Last\_S_{max}$, $Last\_S$, $Last\_T$, $Last\_V$: array of pointers to event;
                        // $Last\_V \equiv$ current least upper bound on source of inconsistency
                        // $Last\_S \equiv$ current least upper bound on consistent cut $S$
last_found: boolean;    // flag to break from the double-nested loop

(3a) for $i = 1$ to $n$ do
(3b)     $last\_found \longleftarrow false$;
(3c)     for $x = first(Slice\_A[i])$ to $last(Slice\_A[i])$ do
(3d)         if $x.operation = receive$ then                     // $\tilde{i}$ is given in $x.partner$
(3e)             for $y = first(Slice\_B[\tilde{i}])$ to $last(Slice\_B[\tilde{i}])$ do
(3f)                 if $x.hash = y.hash$ then
(3g)                     $last\_found \longleftarrow true$; break();   //the break exits loop of line (3e)
(3h)         if $last\_found$ then break();                     //the break exits loop of line (3c)
(3i)     if $last\_found$ then $Last\_S_{max}[i] \longleftarrow \&previous(x)$ else $Last\_S_{max}[i] \longleftarrow \&last(Slice\_A[i])$;
(3j) $Last\_S, Last\_T \longleftarrow Last\_S_{max}$;
(3k) for $i = 1$ to $n$ do
(3l)     $Last\_V[i] \longleftarrow \&last(Slice\_A[i])$;
(3m) repeat
(3n)     for $i = 1$ to $n$ do
(3o)         for $y = next(\star Last\_T[i])$ to $\star Last\_V[i]$ do
(3p)             if $y.operation = send$ then                   // $\tilde{i}$ is given in $y.partner$
(3q)                 for $z = first(Slice\_A[\tilde{i}])$ to $\star Last\_S[\tilde{i}]$ do
(3r)                     if $y.hash = z.hash$ then                // make $Last\_S$ consistent
(3s)                         $Last\_S[\tilde{i}] \longleftarrow \&previous(z)$; break();  // exit loop of (3q)
(3t)     if $Last\_S = Last\_T$ then
(3u)         for $i = 1$ to $n$ do
(3v)             $State\_S[i] \longleftarrow f(State\_Z[i], \langle first(Slice\_A[i]), \ldots \star Last\_S[i]\rangle)$;
(3w)         return();
(3x)     $Last\_V \longleftarrow Last\_T$;
(3y)     $Last\_T \longleftarrow Last\_S$;
(3z) forever.
```

Fig. 6. Iteratively constructing a consistent cut.

After $Slice\_A$ and $Slice\_B$ have been reported to P_{init} at the end of Phases II and III, the initiator invokes procedure *Compute_Consistent_Snapshot* given in Fig. 6 to construct a consistent cut S. The algorithm uses two temporary cuts T and V. Cuts S_{max}, S, T, and V are not explicitly stored as sets but are implicitly identified by vectors $Last\_S_{max}$, $Last\_S$, $Last\_T$, and $Last\_V$ of pointer variables. The ith element (where $1 \leq i \leq n$) of any of these vectors points to the last event at process P_i in the corresponding cut.

$Slice\_A[i]$ and $Slice\_B[i]$, which are sequences of events, are represented as doubly linked lists of type **event** (see Fig. 7). The first event is preceded by a **head** and the last event is succeeded by a **tail**. The operations *first*, *last*, *next*, and *previous* with the standard semantics on these lists are assumed. The vectors $Last\_S_{max}$, $Last\_S$, $Last\_T$, and $Last\_V$ of pointer variables point within the doubly linked lists of $Slice\_A$. See Fig. 7.

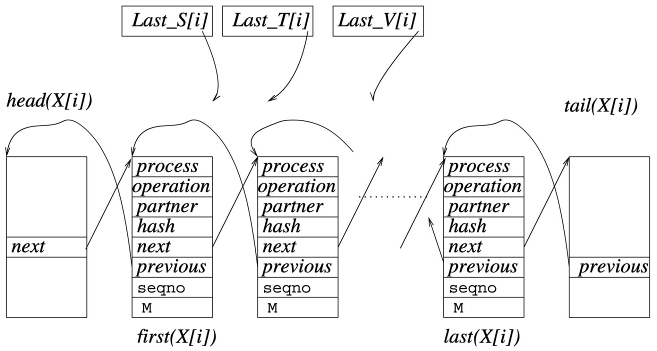

Fig. 7. Implementation of event sequence $X[i]$ (e.g., $Slice\_A[i]$ and $Slice\_B[i]$) as a doubly linked list. The fields in the typewriter font are present only if channel states are to be recorded.

In Fig. 6, lines 3a through 3i construct S_{max} by removing the minimum suffix from $Slice\_A$, in which messages from $Slice\_B$ are received. The solution cut S is then constructed by iteratively removing the minimum slice suffix from cut S_{max} to get a consistent cut. Line 3j initializes the array of pointer variables $Last\_S$ and $Last\_T$ to $Last\_S_{max}$.

- Array $Last\_S$ is always set to the best known least upper bound of the consistent cut S that is sought.
- Array $Last\_V$ denotes the best known least upper bound on the cut V such that *only* messages sent in the slice $V \setminus S$ may cause S to be inconsistent. $Last\_V[i]$ (for $1 \leq i \leq n$) is initialized so as to point to the last event in $Slice\_A$ (lines 3k and 3l).
- Array $Last\_T$ is a temporary variable that is used to update $Last\_V$ at the end of an iteration to the value held by $Last\_S$ at the start of that iteration.

The goal is to construct S such that S is consistent.

The main **repeat** loop (lines 3m to 3z) updates S and V iteratively. In each iteration, it examines each event y in the slice $V \setminus S$ to determine if a message sent at that event was received at an event z that belongs to slice S (lines 3n to 3r). A message at the sender is matched with the same message at the receiver by comparing their hashes/checksums

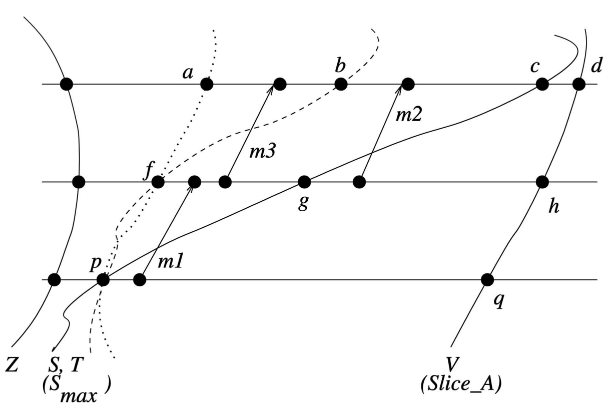

Fig. 8. Example showing removal of inconsistencies.

(line 3r). When such an event z at some process $P_{\bar{i}}$ is identified, the algorithm updates slice S by deleting the suffix at $P_{\bar{i}}$ from event z onward (line 3s). This removes the inconsistency in slice S caused by the message from y to z.

Toward the end of one iteration of the **repeat** loop, V is updated to T (line 3x), and then T is updated to S (line 3y) for the next iteration. Observe that S may still be inconsistent because a message sent in $V \setminus S$ may be received in S. Hence, the **repeat** loop must be executed again unless $S = T$ (line 3t), i.e., no inconsistent-causing message from $V \setminus S$ to S was found in the current iteration. The global state *State_S* is then computed as shown in lines 3t to 3v. The function f begins with event $first(Slice\_A[i])$ for all i, and reruns the computation slice

$$\langle first(Slice\_A[i]) \ldots \star\, Last\_S[i]) \rangle$$

from $State\_Z[i]$ to construct $State\_S$. The rerun of the partial slice is achieved by applying the transformations in the $operation$ fields of noncommunication events in the sequence of events above.

In the code, if the *previous* operation on an **event** in line 3i points to the **head** of the sequence, then

- in line 3o, $next(\star Last\_T[i])$ points to the first event in that sequence, if any, or else to the tail, and
- in line 3q, $Last\_S[i]$ points to the head of that same sequence and the loop does not execute.

Similarly, in line 3s, if $previous(z)$ points to

$$head(Slice\_A[j]),$$

then, in line 3v, $\star Slice\_S[j]$ is the head of a sequence and $State\_S[j]$ is simply set to $State\_Z[j]$.

Example. In Fig. 8, assume the events pointed to by $Last\_S$, $Last\_T$ are $[c, g, p]$, as shown by regular curves. Assume that the events pointed to by $Last\_V$ are $[d, h, q]$. The initial values of $Last\_S, Last\_T$ and of $Last\_V$ are shown in parentheses.

Iteration 1 of repeat loop. After line 3s, $\star Last\_S = [b, f, p]$, $\star Last\_T = [c, g, p]$, and $\star Last\_V = [d, h, q]$. After line 3y, $\star Last\_S$, $\star Last\_T = [b, f, p]$, and $\star Last\_V = [c, g, p]$.

Iteration 2 of repeat loop. After line 3s, $\star Last\_S = [a, f, p]$, $\star Last\_T = [b, f, p]$, and $\star Last\_V = [c, g, p]$. After line 3y, $\star Last\_S, \star Last\_T = [a, f, p]$, and $\star Last\_V = [b, f, p]$.

Iteration 3 of repeat loop. In line 3t, $\star Last\_S = \star Last\_T = [a, f, p]$. So, there is no inconsistency in S and the algorithm exits from the loop.

The correctness of the recorded global state is based on the following theorems proved in the Appendix.

Lemma 1. *In each iteration of the* ***repeat*** *loop of lines 3m to 3z of Compute_Consistent_Snapshot in Fig. 6, only the minimum suffix from the cut identified by Last_S that is needed to remove inconsistency-causing messages is deleted.*

Lemma 2. *For any inconsistency-causing chain M given by $\langle m^1, m^2, \ldots m^k \rangle$, after j iterations of the* ***repeat*** *loop in lines 3m to 3z in Fig. 6, the chain becomes M' given by $\langle m^{j+1} \ldots m^k \rangle$, where $s^{j+1} \notin Last\_S$.*

Lemma 3. *The longest inconsistency-causing chain has length $n - 1$.*

Theorem 1 (Termination). *Procedure Compute_Consistent_Snapshot in Fig. 6 terminates in n iterations of the* ***repeat*** *loop of lines 3m-3z.*

Theorem 2 (Correctness of Local Recordings). *The algorithm initiated by P_{init} terminates by finding a consistent cut S, where $S(t_{A\_start}) \subseteq S \subseteq S_{max}$.*

4 Consistent State and Channel Recording under FIFO Channels

This section presents an enhanced algorithm that also records channel states if channels are FIFO. This algorithm can be used to detect any stable predicate. In the previous algorithm, *Slice_B* was used to construct $Last\_S_{max}$ by identifying those messages received in *Slice_A* that were sent in *Slice_B*. Here, *Slice_B* is also used to capture the channel states. To do so, we require that 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 some extra control information distributively sent to the initiator in the Phase II reply messages and then conveyed on the Phase III request received from the initiator.

Fig. 9 shows the three-phase processing. The underlined pseudocode and data structures mark the differences from the algorithm in Fig. 5. Procedure *Compute_Consistent_Snapshot* in Fig. 6 constructs a consistent cut from *Slice_A* and *Slice_B*, as described in Section 3.

The *state of channel* $C_{i,j}$ after events $\star Last\_S[i]$ and $\star Last\_S[j]$, is defined as the set of messages sent by P_i up to event $\star Last\_S[i]$ that are not received until event $\star Last\_S[j]$ at P_j (and hence received later in *Slice_A* and *Slice_B*). Procedure *Compute_In_Transit_Messages* in Fig. 10 computes the channel states $transit(\star Last\_S[i], \star Last\_S[j])$, for all $\langle i, j \rangle$ pairs, using

- 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, and
- integer vectors $Received[1 \ldots n]$, $Sent[1 \ldots n]$, and

  $$Must\_Receive[1 \ldots n]$$

```
(variables at an initiator)
$State\_Z[1 \dots n]$: array of states;                                  // Phase 1 state recording, stored at initiator
$State\_S[1 \dots n]$: array of states;                                  // states in the solution
$Last\_S[1 \dots n]$: array of pointers to event;                        // pointers to last events in solution cut
$Slice\_A[1 \dots n]$: array of sequence of event;                       // Phase 2 slice recordings, stored at initiator
$Slice\_B[1 \dots n]$: array of sequence of event;                       // Phase 3 slice recordings, stored at initiator
$\underline{Global\_Sent[1 \dots n, 1 \dots n]\text{: array of integer;}}$   //$Global\_Sent[i,j] \equiv$ # messages sent by $P_i$ to $P_j$
$\underline{Global\_Received[1 \dots n, 1 \dots n]\text{: array of integer;}}$
                                           //$Global\_Received[i,j] \equiv$ # messages received by $P_i$ from $P_j$
(variables at each process)
$Slice\_Log$: sequence of event;                                         // log of local events within a slice
$\underline{Sent[1 \dots n]\text{: array of integer;}}$                  // $Sent[k] \equiv$ # messages sent to $P_k$
$\underline{Received[1 \dots n]\text{: array of integer;}}$              // $Received[k] \equiv$ # messages received from $P_k$
$\underline{Must\_Receive[1 \dots n]\text{: array of integer;}}$
                        // $Must\_Receive[k] \equiv$ # 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 $Phase\_1\_Report(state_j, \underline{Received_j[1 \dots n]})$ from each process $P_j$;
(1c)      $(\forall j)\ State\_Z[j] \longleftarrow state_j$ received from $P_j$;
(1c')     $\underline{(\forall j)\ Global\_Received[j][1 \dots n] \longleftarrow Received_j[1 \dots n];}$
(1d) send $Request\_Phase\_2\_Report$ to all $P_j$;
(1e) await $Phase\_2\_Report(Slice\_Log_j, \underline{Sent_j[1 \dots n]})$ from each process $P_j$;
(1f)      $(\forall j)\ Slice\_A[j] \longleftarrow Slice\_Log_j$ received from $P_j$;
(1f')     $\underline{(\forall j)\ Global\_Sent[j][1 \dots n] \longleftarrow Sent_j[1 \dots n] \text{ received from } P_j;}$
(1g) send $Request\_Phase\_3\_Report$ $\underline{\text{with } Global\_Sent[1 \dots n][j] \text{ piggybacked on it}}$ to all $P_j$;
(1h) await $Phase\_3\_Report(Slice\_Log_j)$ from each process $P_j$;
(1i)      $(\forall j)\ Slice\_B[j] \longleftarrow Slice\_Log_j$ received from $P_j$;
(1j) $Compute\_Consistent\_Snapshot(Slice\_A, Slice\_B)$                   // snapshot computed in $State\_S$
(1k) $\underline{Compute\_In\text{-}transit\_Messages(Last\_S)}$.

(2) Process $P_j$ executes the following, for each $j$ such that $1 \leq j \leq n$.
(2a) On receiving $Request\_Phase\_1\_Report$ from $P_{init}$,
(2b)      send $Phase\_1\_Report(state_j, \underline{Received[1 \dots n]})$ to $P_{init}$;
(2c)      Begin recording sequence of events in $Slice\_Log$;
(2d) On receiving $Request\_Phase\_2\_Report$ from $P_{init}$,
(2e)      send $Phase\_2\_Report(Slice\_Log, \underline{Sent[1 \dots n]}))$ to $P_{init}$;
(2f)      Reset $Slice\_Log$; Begin recording sequence of events in $Slice\_Log$;
(2g) On receiving $Request\_Phase\_3\_Report$ $\underline{\text{with } Must\_Receive[1 \dots n] \text{ piggybacked on it}}$ from $P_{init}$,
(2g')     $\underline{\text{Await until, } (\forall k), Received[k] \geq Must\_Receive[k];}$
(2h)      send $Phase\_3\_Report(Slice\_Log)$ to $P_{init}$;
(2i)      Stop recording events in $Slice\_Log$.
```

Fig. 9. Three-phase algorithm to record a global snapshot and channel states. Continued in Fig. 10.

```
$\underline{\text{(4) Process } P_{init} \text{ executes } Compute\_In\text{-}transit\_Messages(Last\_S).}$
(4a) $\forall i \forall j,\ transit(\star Last\_S[i], \star Last\_S[j]) \longleftarrow \emptyset$;
(4b) for $i = 1$ to $n$ do
(4c)      for $x = last(Slice\_A[i])$ down to $next(\star Last\_S[i])$ do
(4d)           if $x.operation = send$ then                                  // $\tilde{i}$ is given in $x.partner$
(4e)                $Global\_Sent[i, \tilde{i}] - -$;
(4f) for $j = 1$ to $n$ do
(4g)      for $x = first(Slice\_A[j])$ to $last(Slice\_B[j])$ do
(4h)           if $x.operation = receive$ then                               // $\tilde{j}$ is given in $x.partner$
(4i)                $Global\_Received[j, \tilde{j}] + +$;
(4j)                if $Global\_Sent[\tilde{j}, j] \geq Global\_Received[j, \tilde{j}]$ and $x.seqno > \star Last\_S[j].seqno$ then
(4k)                     $transit(\star Last\_S[\tilde{j}], \star Last\_S[j]) \longleftarrow transit(\star Last\_S[\tilde{j}], \star Last\_S[j]) \cup \{x.M\}$
```

Fig. 10. Computing in-transit messages.

at each node. $Sent[j]$ and $Received[j]$ track the number of messages sent to and received from process P_j, respectively.

One important difference from the previous algorithm is that sequence numbers are used to count events at a process. This is because for a message received within $Slice\_B$, it becomes necessary to identify whether the message was sent before $Slice\_A$ or after $Slice\_B$. To see this further, consider Fig. 11. Sequence numbers within each local execution can help to determine whether $m1$ and $m2$ were sent before or after the three-phase recordings. To compute the channel state while satisfying conditions P1 through 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 event $\star Last\_S[i]$ must have reached P_j by $last(Slice\_B[j])$.

   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

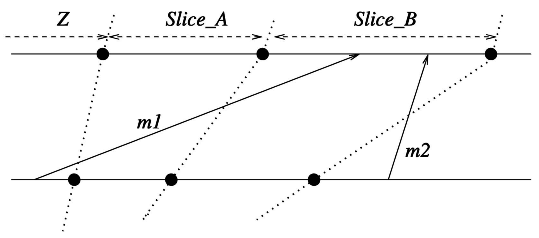

Fig. 11. Pre-$Slice\_A$ and post-$Slice\_B$ message send events need to be distinguished from each other.

$Global\_Sent$ (line 1f′). 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 1g). A process postpones the end of the recording of the local log of $Slice\_B$ until all these number of messages, remembered in array $Must\_Receive$, have been delivered locally (line 2g′).

2. The set of messages sent by P_i to P_j up to $\star Last\_S[i]$, denoted as $\mathcal{X}$, should be identifiable.

   This set contains all the messages sent by P_i (to P_j) with sequence numbers less than or equal to the value of $Sent[j]$ after $\star Last\_S[i]$. This value is computed (lines 4b-4e) using $Global\_Sent$ and $Slice\_A$ that are reported in Phase II. The resulting message count is stored in situ in $Global\_Sent[i,j]$.

3. The set of messages received by P_j from P_i after $\star Last\_S[j]$, denoted as $\mathcal{Y}$, should be identifiable.

   The messages received by P_j are enumerated in lines (4f-4k) using *Global_Received*, *Slice_A*, and $Slice\_B$. The enumeration is done dynamically in situ in $Global\_Received[j,i]$. Among these messages received by P_j, those that are received after $\star Last\_S[j]$ must be received at events x satisfying the condition $x.seqno > \star Last\_S[j].seqno$.

From the above and the definition of *transit*, we have

$$transit(\star Last\_S[i], \star Last\_S[j]) = \mathcal{X} \cap \mathcal{Y} \qquad (1)$$

which is computed in line 4k:

$$\{M\ received\ by\ P_j\ at\ event\ x \mid (Global\_Received[j,i]\ at\ x) \leq Global\_Sent[i,j] \bigwedge x.seqno > \star Last\_S[j].seqno\}. \qquad (2)$$

The correctness proof of the recorded channel states is based on the following theorem, whose proof is given in the Appendix.

Theorem 3. (Correctness of Channel States). *The algorithm initiated by P_{init} correctly computes channel states using procedure Compute_In-transit_Messages when assuming FIFO channels and using local message counters.*

5 Complexity

A key feature of our algorithm is that it does not rely on the processes continually and pessimistically reporting their (relevant) activity. The algorithm can be invoked at any arbitrary time, and only the (relevant) activity that occurs in the slice during the algorithm execution needs to be reported. The size of a slice is an important parameter in the complexity of the algorithm. Before embarking on the complexity analysis, we note that *Slice_A* and *Slice_B* are both *thin* slices. We define

- $\widehat{rtt}_{max}$ to be the expected *r*ound-*t*rip *t*ransit time between the two farthest nodes in the network.

The expected size of *Slice_A* and *Slice_B* is the number of (relevant) events in the execution logs that occur in a duration of time $\widehat{rtt}_{max}$. With current Internet technology, the round-trip times even across the globe are up to 200 ms. On the other hand, a round-trip communication on an MIMD distributed memory message-passing multiprocessor (e.g., IBM Blue Gene, Cray, Fujitsu, and Hitachi series) using MPI takes microseconds [27].

The following complexity analysis assumes a flat tree overlay topology with the initiator as the root. A similar analysis can be conducted for the ring and more general tree topologies.

Message complexity. From Fig. 2, it is seen that $6(n-1)$ messages are required across the three phases.

- The total space requirement across all the messages is $O(|Slice\_A| + |Slice\_B|)$.
- If in-transit messages are to be captured, an additional $O(n^2)$ overhead is incurred due to the n $Sent$ and *Received* vectors.

Message time complexity. Assume that it takes one time unit for a message to reach its destination. The three-phase algorithm (assuming the flat tree topology) takes six time units for the messages in transit. Alternately, this can be expressed as $3\widehat{rtt}_{max}$.

Computation time complexity. The computation time complexity is the execution cost. This analysis uses Fig. 5 (and its variant Fig. 9), Fig. 6, and Fig. 10.

Step 1.

- At the initiator P_{init}, the time complexity is
  $$O(|Slice\_A| + |Slice\_B|).$$
- If in-transit messages are to be reported, an additional $O(n^2)$ overhead is incurred because of arrays *Global_Sent* and *Global_Received*.

Step 2.

- At a noninitiator P_j, the time complexity is
  $$O((|Slice\_A| + |Slice\_B|)/n)$$
  average case.
- If in-transit messages are to be reported, an added $O(n)$ overhead is incurred due to vectors *Sent*, *Received*, and *Must_Receive*.

Step 3: Procedure *Compute_Consistent_Snapshot*.

- Non-FIFO version, no channel state recording.
  - The loop (3a-3i) requires $O(1/n \cdot (|Slice\_A| \cdot |Slice\_B|))$ time. This is because for each process, each of the $1/n \cdot |Slice\_A|$ events may have to be examined if they were receive events, and, if so,

TABLE 3
Complexity of the Snapshot Algorithms, Usable for Stable Property Detection

Metric	Snapshot Algorithm v.1 (no channel states recorded)	Snapshot algorithm v.2 (channel states recorded)
Number of messages; Message space (total); Message time complexity	$6(n-1)$; $O(\|Slice\_A\|)$; 6 hops or $3\widehat{rtt}_{max}$	$6(n-1)$; $O(\|Slice\_A\|)+O(n^2)$; 6 hops or $3\widehat{rtt}_{max}$
Computation time complexity (initiator)	$O(1/n \cdot \|Slice\_A\|^2)$ average $+O(\|Slice\_A\|)$ $+O(n^2)$	$O(1/n \cdot \|Slice\_A\|^2)$ average $+O(\|Slice\_A\|)$ $+O(n^2)$
Computation time complexity (non-initiator)	$O(\|Slice\_A\|/n)$ average	$O(\|Slice\_A\|/n)$ average $+O(n)$
Space complexity (initiator)	$O(\|Slice\_A\|)$	$O(\|Slice\_A\|)+O(n^2)$
Space complexity (noninitiator)	$O(1/n \cdot \|Slice\_A\|)$ average	$O(1/n \cdot \|Slice\_A\|)$ average $+O(n)$
Features	No inhibition Application messages unmodified execution unmodified no log of history	No inhibition Application messages unmodified execution unmodified no log of history *Sent* & *Received* vectors per process *Global_Sent*, *Global_Receive* arrays at initiator
Channels	non-FIFO	FIFO
Stable properties detected	Locally stable	All

the hash of the message received has to be compared against the hash of the messages sent at each of the possible $(1/n \cdot |Slice\_B|)$ events at the sender process. Thus, for each process execution considered from *Slice_A*, the overhead contributed is $1/n \cdot |Slice\_A| \cdot 1/n \cdot |Slice\_B|$. Across all processes, the overhead becomes $O(1/n \cdot |Slice\_A| \cdot |Slice\_B|)$ average case.

- Consider the main loop (3m-3z). Observe that each event in *Slice_A* gets examined at most once in line 3o. If a message is sent at this event, due to non-FIFO channels, the hash of this message needs to be compared with the hash of the messages received at possibly *each* of the events, numbering $1/n \cdot |Slice\_A|$, at the receiver process. The overhead across all n iterations for 3o to 3s is $O(1/n \cdot |Slice\_A|^2)$. There is one iteration of lines 3u to 3v, taking time $O(|Slice\_A|)$. There are at most n iterations of lines 3x-3y, costing a total of $O(n^2)$ time. So the time overhead for loop 3m-3z is $O(1/n \cdot |Slice\_A|^2 + |Slice\_A| + n^2)$ average case.

- FIFO version, with recording of channel states
  - For the loop in lines 3a to 3i, the overhead is the same as for the non-FIFO version, $O(1/n \cdot |Slice\_A| \cdot |Slice\_B|)$ average case.
  - For the main loop 3m to 3z, the overhead is the same as that for the non-FIFO version, namely, $O(1/n \cdot |Slice\_A|^2 + |Slice\_A| + n^2)$ average case.

Step 4: Procedure *Compute_In-transit_Messages*.

- Line 4a requires $O(n^2)$ time. Lines 4b to 4e require $O(|Slice\_A|)$ time. The main loop in lines 4f to 4k requires $O(|Slice\_A| + |Slice\_B|)$ time because it requires a single pass through the events of both the slices.

At the initiator P_{init}, the average case computation time complexity for Version 1 (no channel recording) and for Version 2 (channel recording, FIFO channels) is

$$O(|Slice\_A| + |Slice\_B|) + O(1/n \cdot (|Slice\_A| \cdot |Slice\_B|)) + O(1/n \cdot |Slice\_A|^2 + |Slice\_A| + n^2). \tag{3}$$

Space complexity.

- At initiator.
  - The total space complexity is
  $$O(|Slice\_A| + |Slice\_B|).$$
  - If in-transit messages are recorded, $O(n^2)$ additional overhead is incurred for arrays *Global_ Sent* and *Global_Received*.
- At noninitiator:
  - The average space complexity is
  $$(1/n \cdot \max(|Slice\_A|, |Slice\_B|))$$
  for the log of the slice.
  - If channel states are recorded, $O(n)$ additional space is required for the *Sent* and *Received* vectors.

The above analysis considered *Slice_A* and *Slice_B* separately. Note that *Slice_A* and *Slice_B* are thin slices and are expected to have similar size. Assuming $|Slice\_A| = |Slice\_B|$, Table 3 summarizes the complexity results for the proposed algorithm.

6 Detecting Stable Predicates

Theorems 2 and 3 showed the safety of the snapshot algorithm. The proposed algorithm Version 2 to record a consistent global state can be used in a straightforward manner to detect any stable predicate. 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 cut S, it can also evaluate the predicate over these variables after cut S. If the predicate evaluates to *true*, then it is true and remains true henceforth because it is stable. Safety of the predicate detection follows from the correctness of Theorems 2 and 3 and the definition of a stable predicate. A similar argument holds for Version 1 of the algorithm which detects any locally stable predicate. We now analyze the liveness of both versions of the algorithm.

- Version 1 can detect locally stable predicates. A locally stable predicate can be evaluated over the variables at the processes. Liveness follows from the fact that if a locally stable predicate becomes true after some cut Q, then any invocation of the snapshot algorithm which computes a cut S such that $Q \subseteq S$ will detect the locally stable predicate.
- Version 2 can detect any stable predicate. If a stable predicate is true in S, it will be detected because the entire system state (processes and channels) is observable at S. Specifically, liveness follows from the fact that if a stable predicate becomes true in some global state Q, then any invocation of the snapshot algorithm which computes a state S such that $Q \subseteq S$ will detect the stable predicate.

7 Algorithm Variants

Our algorithm V2 used the $Sent$ and $Received$ vectors to compute the channel states in the "Reduce A" approach. This section gives two variants of V2. The first variant V2.A uses the $Sent$ and $Received$ vectors to also construct the consistent cut S. The second variant V2.B uses the "Extend Z" approach to construct the consistent cut. V2.A has somewhat better complexity than V2. V2.B has a higher complexity than V2 but is given to demonstrate the complementary approach to the "Reduce A" approach.

7.1 Reducing the "Second Cut" Using Counters (v.2.A)

Version 2 constructed S using the idea of detecting inconsistencies by tracing through $Slice\_A$ (see Fig. 6). The $Sent$ and $Received$ vectors are required to capture in-transit messages. However, these vectors can also be used for constructing the consistent cut S—the algorithm is very similar to that in Fig. 6, Section 3, and, hence, the repeat loop has at most n iterations. The algorithm is given in Fig. 12. $Global\_Sent[i, \tilde{i}]$ and $Global\_Received[i, \tilde{i}]$ get updated to their values for event $\star Last\_S[i]$, in lines 3A and 3u, respectively. Hence, Step 4 gets simplified from that in Fig. 10.

The $Global\_Received$ is initialized at the end of Phase I, whereas the $Global\_Sent$ is initialized at the end of Phase II. In lines 3a to 3g, S_{max} is constructed and $Global\_Received$ is modified (line 3e) to reflect the receive count at S_{max}. Lines 3h to 3k modify $Global\_Sent$ to reflect the send count at S_{max}. Lines 3l to 3n initialize $Last\_T$, $Last\_V$, and $Last\_S$. The main **repeat** loop (3o-3H) achieves the counterpart of lines 3m to 3z of Fig. 6. As the code iterates to construct S from S_{max}, so also in each iteration, $Global\_Received$ is decremented in line 3u to reflect the receive count of S for that iteration of the **repeat** loop. Analogously, the $Global\_Sent$ count is decremented in line 3A to reflect the send count of S in that iteration of the **repeat** loop.

Complexity. The complexity for all the metrics except the computation time complexity is the same as for the algorithm v.2 in Fig. 6 (see Table 3). Lines 3a to 3g construct S_{max}—the initiator can detect a message received at an event in $Slice\_A$ as being sent from an event in $Slice\_B$ in $O(|Slice\_A|)$ time. Lines 3h-3k which update $Global\_Sent$ for this S_{max} also take $O(|Slice\_A|)$ time. Across all iterations of the **repeat** loop, the number of times event y is enumerated in line 3q is at most $|Slice\_A|$, and the number of times event z is enumerated in line 3s is at most $|Slice\_A|$. Similarly, across all iterations of the **repeat** loop, the number of times event y is enumerated in line 3y is at most $|Slice\_A|$. The lines 3B-3D are executed once, leading to $|Slice\_A|$ overhead. Lines 3F-3G contribute $O(n^2)$ overhead across the maximum of n executions of the **repeat** loop. Hence, the time complexity of Step 3 is $O(|Slice\_A| + n^2)$. The time complexity of Step (4) is $O(|Slice\_A| + |Slice\_B|)$. Assuming $|Slice\_A| = |Slice\_B|$, the time complexity of the algorithm is $O(|Slice\_A| + n^2)$.

7.2 Extending the "First Cut" (v.2.B)

Using the $Sent$ and $Received$ vectors that are required to capture the in-transit messages, it is possible to construct the consistent cut S using the "extend Z" approach described in Section 3.1.

- Steps 1 and 2 will differ from Fig. 9 as follows: The $Sent$ vectors will be reported in the Phase I reply (i.e., for cut Z) instead of in the Phase II reply.
- The modified Step 3 is described in Fig. 13. Step 4 is the same as in Fig. 12.

In Step 3, cuts S and V are initialized to Z in lines 3a to 3b. Each iteration of the **repeat** loop adds the minimum prefix from $Slice\_A \setminus S$ to S, to remove any inconsistency by including necessary send events (lines 3d to 3i). However, when such a prefix is added to S, some receive events may also be included such that the corresponding send events are not included in the prefixes added in this iteration. Thus, further iterations of the **repeat** loop are needed until S does not change in an entire iteration (line 3n). $Last\_V$ is a working variable used to track the current greatest lower bound on the source of inconsistency in an iteration. $Global\_Sent[i, \tilde{i}]$ and $Global\_Received[i, \tilde{i}]$ get updated to their values for event $\star Last\_S[i]$, in lines 3h and 3m, respectively. Hence, Step 4 gets simplified from that in Fig. 10, and is the same as shown in Fig. 12. The correctness proof is analogous to that for the algorithm presented in Sections 3 and 4.

Complexity. For the "Extend Z" approach, the complexity for all the metrics except the computation time complexity is the same as for the "Reduce A" approach (see Algorithm v.2 in Table 3).

It can be seen that there are at most n iterations of the main **repeat** loop in Fig. 13. In each iteration,

- $Slice\_A$ is examined in lines 3d to 3i, leading to $O(|Slice\_A|)$ time overhead.
- Lines 3n and 3r contribute $O(n)$ time overhead.

```
(3) Process $P_{init}$ executes $Compute\_Consistent\_Snapshot$.
$Last\_S_{max}$, $Last\_S$, $Last\_T$, $Last\_V$: array of pointers to event;
$found$: boolean;                                   // latest event to construct $S_{max}$ at a process

(3a) for $i = 1$ to $n$ do
(3b)     for $x = first(Slice\_A[i])$ to $last(Slice\_A[i])$ do
(3c)         if $x.operation = receive$ then                          // $\tilde{i}$ is given in $x.partner$
(3d)             if $Global\_Received[i, \tilde{i}] < Global\_Sent[\tilde{i}, i]$ then   // update $Global\_Received$ for $S_{max}$
(3e)                 $Global\_Received[i, \tilde{i}] + +$;
(3f)             else $found \longleftarrow true$; break();                    // exit loop of line (3b)
(3g)     if $found$ then $Last\_S_{max}[i] \longleftarrow \&previous(x)$ else $Last\_S_{max}[i] \longleftarrow \&last(Slice\_A[i])$;
(3h) for $i = 1$ to $n$ do                                    // constructed $S_{max}$ in (3g)
(3i)     for $x = last(Slice\_A[i])$ down to $next(\star Last\_S_{max}[i])$ do
(3j)         if $x.operation = send$ then                         // $\tilde{i}$ is given in $x.partner$
(3k)             $Global\_Sent[i, \tilde{i}] - -$;                         // update $Global\_Sent$ for $S_{max}$
(3l) $Last\_S, Last\_T \longleftarrow Last\_S_{max}$;
(3m) for $i = 1$ to $n$ do
(3n)     $Last\_V[i] \longleftarrow \&last(Slice\_A[i])$;
(3o) repeat
(3p)     for $i = 1$ to $n$ do
(3q)         for $y = next(\star Last\_T[i])$ to $\star Last\_V[i]$ do             // $\tilde{i}$ is given in $y.partner$
(3r)             if $y.operation = send$ and $Global\_Sent[i, \tilde{i}] < Global\_Received[\tilde{i}, i]$ then
(3s)                 for $z = \star Last\_S[\tilde{i}]$ down to $first(Slice\_A[\tilde{i}])$ do
(3t)                     if $z.operation = receive$ then              // $\tilde{\tilde{i}}$ is given in $z.partner$
(3u)                         $Global\_Received[\tilde{i}, \tilde{\tilde{i}}] - -$;  //adjust $Global\_Received$ for $S$
(3v)                         if $Global\_Sent[i, \tilde{i}] \geq Global\_Received[\tilde{i}, i]$ then
(3w)                             $Last\_S[\tilde{i}] \longleftarrow \&previous(z)$; break();  // exit loop (3s)
(3x)     for $i = 1$ to $n$ do
(3y)         for $y = \star Last\_T[i]$ down to $next(\star Last\_S[i])$ do
(3z)             if $y.operation = send$ then                      // $\tilde{i}$ is given in $y.partner$
(3A)                 $Global\_Sent[i, \tilde{i}] - -$;                    // update $Global\_Sent$ for $S$
(3B)     if $Last\_S = Last\_T$ then
(3C)         for $i = 1$ to $n$ do
(3D)             $State\_S[i] \longleftarrow f(State\_Z[i], \langle first(Slice\_A[i]), \ldots \star Last\_S[i] \rangle)$;
(3E)         return();
(3F)     $Last\_V \longleftarrow Last\_T$;
(3G)     $Last\_T \longleftarrow Last\_S$;
(3H) forever.

(4) Process $P_{init}$ executes $Compute\_In\text{-}transit\_Messages(Last\_S)$.
(4a) $\forall i \forall j$, $transit(\star Last\_S[i], \star Last\_S[j]) \longleftarrow \emptyset$;
(4b) for $j = 1$ to $n$ do
(4c)     for $x = next(\star Last\_S[j])$ to $last(Slice\_B[j])$ do
(4d)         if $x.operation = receive$ then                      // $\tilde{j}$ is given in $x.partner$
(4e)             $Global\_Received[j, \tilde{j}] + +$;
(4f)             if $Global\_Sent[\tilde{j}, j] \geq Global\_Received[j, \tilde{j}]$ then
(4g)                 $transit(\star Last\_S[\tilde{j}], \star Last\_S[j]) \longleftarrow transit(\star Last\_S[\tilde{j}], \star Last\_S[j]) \cup \{x.M\}$
```

Fig. 12. Version V2A: Constructing consistent cut S for the Version 2 (FIFO channels) algorithm using *Global_Sent* and *Global_Received.*

Lines 3o to 3q, which are executed only once, contribute $O(|Slice\_A|)$ time overhead. Lines 3j to 3m contribute $O(|Slice\_A|)$ time overhead across all iterations. So the execution time complexity at the initiator is

$$O(n \cdot |Slice\_A| + n^2).$$

7.3 Complexity Trade-offs

The complexity of the three snapshot algorithms—"Reduce A" without arrays, (Fig. 6), "Reduce A" with arrays (Fig. 12), and "Extend Z" with arrays, (Fig. 13)—is the same for all metrics except the time computation complexity at the initiator. This computation time complexity is compared in Table 4. The $O(n^2)$ is a lower bound because the states of $O(n^2)$ channels have to be recorded.

Let w, termed the *slice width*, be the average number of events at a process in a slice, i.e., $w = |Slice\_A|/n$. The width w depends on the geographical span of the network as well as the expected rate of generation of relevant events. For *thin slices* and large systems, it is likely that $w < n$. The computation time complexity of the algorithms in terms of w is also given in Table 4. In order to remain bounded by $O(n^2)$, the relationship between $O(w)$ and $O(n)$ is also expressed. The algorithm in Fig. 12 provides the best flexibility. The algorithm using the "Extend Z" approach has higher complexity but is given for completeness as it complements the "Reduce A" approach.

```
(3) Process $P_{init}$ executes $Compute\_Consistent\_Snapshot$.
$Last\_S$, $Last\_V$: array of pointers to event;
                         //$Last\_V \equiv$ current greatest lower bound on source of inconsistency
                         //$Last\_S \equiv$ current greatest lower bound on consistent cut $S$

(3a) for $i = 1$ to $n$ do
(3b)     $Last\_S[i], Last\_V[i] \longleftarrow \&head(Slice\_A[i])$;
(3c) repeat
(3d)     for $i = 1$ to $n$ do
(3e)         for $y = next(\star Last\_V[i])$ to $last(Slice\_A[i])$ do
(3f)             if $y.operation = send$ then                     // $\tilde{i}$ is given in $y.partner$
(3g)                 if $Global\_Sent[i,\tilde{i}] < Global\_Received[\tilde{i}, i]$ then   // make $S$ consistent
(3h)                     $Global\_Sent[i,\tilde{i}]++$;
(3i)                     $Last\_S[i] \longleftarrow \&y$;
(3j)     for $i = 1$ to $n$ do
(3k)         for $y = next(\star Last\_V[i])$ to $\star Last\_S[i]$ do
(3l)             if $y.operation = receive$ then                  // $\tilde{i}$ is given in $y.partner$
(3m)                 $Global\_Received[i,\tilde{i}]++$;      // update $Global\_Received$ for new $S$
(3n)     if $Last\_S = Last\_V$ then
(3o)         for $i = 1$ to $n$ do
(3p)             $State\_S[i] \longleftarrow f(State\_Z[i], \langle first(Slice\_A[i]), \ldots \star Last\_S[i]\rangle)$;
(3q)         return();
(3r)     $Last\_V \longleftarrow Last\_S$;
(3s) forever.
```

Fig. 13. Iteratively constructing a consistent cut using the "Extend Z" approach.

TABLE 4
Computation Time Complexity of the Snapshot Algorithms That Record States of FIFO Channels

Snapshot Algorithm v.2 ("Reduce A", Figure 6)	Snapshot algorithm v.2.A ("Reduce A", Figure 12)	Snapshot algorithm v.2.B ("Extend Z", Figure 13)
$O(1/n \cdot \lvert Slice\_A\rvert^2) + O(\lvert Slice\_A\rvert) + O(n^2)$	$O(\lvert Slice\_A\rvert) + O(n^2)$	$O(n \cdot \lvert Slice\_A\rvert) + O(n^2)$
$= O(1/n \cdot (nw)^2) + O(nw) + O(n^2)$	$= O(nw) + O(n^2)$	$= O(n \cdot nw\rvert) + O(n^2)$
$= O(nw^2 + n^2)$	$= O(nw + n^2)$	$= O(n^2 w + n^2)$
$= O(n^2)$ if $O(w) < O(\sqrt{n})$	$= O(n^2)$ if $O(w) < O(n)$	$= O(n^2 w)$

7.4 Two-Phase Algorithm: Local States under FIFO Channels, V2.C

Observe that the modified code in Fig. 12 to construct S_{max} using $Global\_Sent$ and $Global\_Received$ does not use $Slice\_B$. This immediately suggests a two-phase algorithm to record local states consistently if each process maintains the $Sent$ and $Received$ counters. The two-phase exchange would be as shown in Fig. 9—only lines 1g to 1i and 2f to 2h would be omitted. The cost would be the same as for the snapshot algorithm v.2 (see Table 3) except that

- Number of messages $= 4(n-1)$, message time complexity is four hops or $2\widehat{rtt}_{max}$ and locally stable properties are detected.

8 Discussion

This paper first presented two algorithms to detect stable properties. The first algorithm can be used to detect any locally stable properties in a non-FIFO system, whereas the second algorithm can detect any stable property in a FIFO system, under the assumptions P1-P4, Section 1. A simple and elegant three-phase strategy of uncoordinated observation of local states was used to create a consistent distributed snapshot. The first snapshot algorithm computed consistent process states without requiring FIFO channels. The second algorithm computed process states and channel states consistently but required FIFO channels. A key feature of our algorithms is that they do not rely on the processes continually and pessimistically reporting their (relevant) activity. The algorithms can be invoked at any arbitrary time, and only the (relevant) activity that occurs in the *thin* slice during the algorithm execution needs to be reported. The paper then showed three variants of the algorithms, that adopt a complementary approach to the originally presented algorithms.

The goal of procedure *Compute_Consistent_Snapshot* resembles the goal of finding a consistent rollback state when using asynchronous checkpointing [24]. The idea in both is to construct a consistent state out of an inconsistent state, given the logs of the execution. However, the approaches, algorithms, and purposes are different. Further, our approach also does not require logging the entire history of the execution, does not modify application messages by tagging sequence numbers, and does not use a reactive event model for the execution.

The snapshot algorithms presented here create an on-demand *thin slice* of the execution that is guaranteed to contain a consistent cut. The "thin slice" approach involves very low overhead for the algorithm when channel states are not needed. The proposed snapshot algorithm that also computes channel states requires $O(n^2)$ time and space at the initiator, which can be viewed as a drawback. However, note that in practice, this is preferable to existing algorithms that require $O(n^2)$ messages instead of $O(n)$ messages. The proposed algorithms also have a snapshot readily available at the initiator, whereas most of the existing algorithms record a distributed snapshot and need to incur an additional cost to collect it.

The proposed algorithms record *some* consistent global state in $Slice\_A$. In order to construct a global state that contains a particular local state at the initiator, the following can be done. In the Phase II recordings, the initiator needs to ensure that it records its local state before any other process. (We leave the correctness reasoning to the reader). The initiator still needs to have initiated Phase I earlier, which means that if a look-ahead is not possible, the initiator needs to freeze its execution between its Phase I and Phase II recordings.

APPENDIX A
CORRECTNESS PROOF

A.1 Consistency

Lemma 1. *In each iteration of the* **repeat** *loop of lines 3m to 3z of Compute_Consistent_Snapshot in Fig. 6, only the minimum suffix from the cut identified by Last_S that is needed to remove inconsistency-causing messages is deleted.*

Proof Sketch. This is evident from lines 3n to 3s, where $Last\_S[i]$ is adjusted to point to the earlier of 1) the current event being pointed to and 2) the event immediately preceding the receipt of a message that was sent *after* the event pointed to by $Last\_T[i]$. Note that $Last\_T[i]$ and $Last\_S[i]$ are the same at the start of each iteration; hence any message identified in event 2 causes the state identified by S to be inconsistent. This adjustment in line 3s eliminates the minimum suffix from the local state identified by $Last\_S[i]$ in order to eliminate the receive event and restore consistency (with respect to this inconsistency-causing message). □

Denote a message m^i by its send and receive events $\langle s^i, r^i\rangle$. A message chain is a causal sequence of messages $\langle m^1, m^2 \dots m^k\rangle$ such that $r^i \prec s^{i+1}$, for $i \in [1 \dots (k-1)]$, and events r^i and s^{i+1} must occur at the same process. For simplicity, we denote the cut S by the pointer array $Last\_S$. An inconsistency-causing message chain is such that $s^1 \notin Last\_S$ and $r^1 \in Last\_S$, and, for all $j \in [2 \dots k]$, we also have $s^j, r^j \in Last\_S$.

Lemma 2. *For any inconsistency-causing chain M given by $\langle m^1, m^2, \dots m^k\rangle$, after j iterations of the* **repeat** *loop in lines 3m to 3z in Fig. 6, the chain becomes M' given by $\langle m^{j+1} \dots m^k\rangle$, where $s^{j+1} \notin Last\_S$.*

Proof Sketch. We show this by induction on the number of iterations of the **repeat** loop.

Case $\mathbf{j = 1}$. From Lemma 1, only the minimal inconsistency-causing suffix due to the receive event r^1 is removed. As $r^1 \prec s^2$, we therefore have the chain $\langle m^2, m^3 \dots m^k\rangle$ where $s^2 \notin Last\_S$ and all other send and receive events in the inconsistency-causing chain are in $Last\_S$.

Case $\mathbf{j = x}$. Assume that the hypothesis is true for any value of j. So, after x iterations, we have the inconsistency-causing chain $\langle m^{x+1} \dots m^k\rangle$, where $s^{x+1} \notin Last\_S$ and all other send and receive events in the inconsistency-causing chain are in $Last\_S$.

Case $\mathbf{j = x + 1}$. From the induction hypothesis, we have the inconsistency-causing chain $\langle m^{x+1} \dots m^k\rangle$, where $s^{x+1} \notin Last\_S$ and all other send and receive events in the inconsistency-causing chain are in $Last\_S$. From Lemma 1, only the minimal inconsistency-causing suffix due to the receive event r^{x+1} is removed. As $r^{x+1} \prec s^{x+2}$, we therefore have the inconsistency-causing chain

$$\langle m^{x+2}, m^{x+3} \dots m^k\rangle,$$

where $s^{x+2} \notin Last\_S$ and all other send and receive events in the inconsistency-causing chain are in $Last\_S$.□

Lemma 3. *The longest inconsistency-causing chain has length $n - 1$.*

Proof Sketch. We prove by contradiction. Let the last message in the chain be m^k, $k > n - 1$. As the longest inconsistency-causing chain has length n or greater than n, then there is at least one process that has a receive event r^k that happens later than s^h at the same process, i.e., m^h was sent earlier than receiving m^k. As the send event s^h happened before r^k and s^h itself did not belong to the cut marked by $Last\_S$ in iteration h, we have that r^k also cannot belong to the cut $Last\_S$. Hence, m^k does not satisfy the condition of being a part of an inconsistency-causing chain. □

Theorem 1. *Procedure Compute_Consistent_Snapshot in Fig. 6, terminates in n iterations of the repeat loop of lines 3m to 3z.*

Proof Sketch. Follows from Lemmas 1, 2, and 3. The longest inconsistency-causing chain has $n - 1$ messages. After $n - 1$ iterations, there is no inconsistency-causing chain. An additional iteration is required to detect that $Last\_T = Last\_S$ (line 3t) and then to exit the **repeat** loop. □

Theorem 2 (Correctness of Local Recordings). *The algorithm initiated by P_{init} terminates by finding a consistent cut S, where $S(t_{A\_start}) \subseteq S \subseteq S_{max}$.*

Proof Sketch. Steps 1 and 2 of the algorithm contain a three-phase exchange, which can be seen to terminate. Specifically, the only waiting is for sent messages to be received, and assuming a reliable system, this waiting is finite in an asynchronous system. From Theorem 1, Step 3, *Compute_Consistent_Snapshot*, also terminates. Step 4, *Compute_In-transit_Messages*, does local processing and can be seen to terminate. Hence, the algorithm terminates.

The *Compute_Consistent_Snapshot* procedure initializes S to S_{max}. The only operations on S in each iteration are the deletion of a suffix. Hence, on termination, $S \subseteq S_{max}$.

There are a bounded number of events in $S_{max} \setminus S(t_{A\_start})$ and in each iteration that does not result in termination, at least one event is removed. From Lemma 1, only the minimum suffix that causes inconsistency is deleted at each step. Hence, in a finite number of steps, either $S(t_{A\_start})$ or some cut S, where $S(t_{A\_start}) \subset S$, is constructed such that no process would have a suffix deleted. This would be a consistent cut. The theorem now follows. □

A.2 Recording Channel States

Theorem 3 (Correctness of Channel States). *The algorithm initiated by P_{init} correctly computes channel states using procedure Compute_In-transit_Messages when assuming FIFO channels and using local message counters.*

Proof Sketch. For channel $C_{i,j}$, the in-transit messages are those sent up to event $\star Last\_S[i]$ and received after event $\star Last\_S[j]$. Observe from Step 2g′ of Fig. 9 that all messages sent up to $\star Last\_S[i]$ are guaranteed to be received by $last(Slice\_B[i])$, if channels are FIFO. Observe from loop 4f-4k that the channel state is computed as per (2). All messages sent up to $\star Last\_S[i]$ and received after $\star Last\_S[j]$ are therefore identified as belonging to the channel state using the local message counters. □

Acknowledgments

The authors thank the referees for their valuable comments that have significantly simplified the organization and presentation of the results. A preliminary conference version of this result appears in [13].

References

[1] A. Acharya and B.R. Badrinath, "Recording Distributed Snapshots Based on Causal Order of Message Delivery," *Information Processing Letters,* vol. 44, no. 6, pp. 317-321, 1992.
[2] S. Alagar and S. Venkatesan, "An Optimal Algorithm for Distributed Snapshots with Causal Message Ordering," *Information Processing Letters,* vol. 50, no. 6, pp. 311-316, 1994.
[3] R. Atreya, N. Mittal, and V.K. Garg, "Detecting Locally Stable Predicates without Modifying Application Messages," *Proc. Int'l Conf. Principles of Distributed Systems,* pp. 20-33, 2003.
[4] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Trans. Computer Systems,* vol. 3, no. 1, pp. 63-75, 1985.
[5] C. Critchlow and K. Taylor, "The Inhibition Spectrum and the Achievement of Causal Consistency," *Distributed Computing,* vol. 10, no. 1, pp. 11-27, 1996.
[6] C.J. Fidge, "Logical Time in Distributed Computing Systems," *Computer,* vol. 24, no. 8, pp. 28-33, 1991.
[7] J.-M. Helary, "Observing Global States of Asynchronous Distributed Applications," *Proc. Int'l Workshop Distributed Algorithms,* pp. 124-125, 1989.
[8] J. Helary, C. Jard, N. Plouzeau, and M. Raynal, "Detection of Stable Properties in Distributed Applications," *Proc. ACM Symp. Principles of Distributed Computing,* pp. 125-136, 1987.
[9] J. Helary and M. Raynal, "Towards the Construction of Distributed Detection Programs, with an Application to Distributed Termination," *Distributed Computing,* vol. 7, pp. 137-147, 1994.
[10] G.S. Ho and C.V. Ramamoorthy, "Protocols for Deadlock Detection in Distributed Database Systems," *IEEE Trans. Software Eng.,* vol. 8, no. 6, pp. 554-557, 1982.
[11] A. Kshemkalyani, M. Raynal, and M. Singhal, "An Introduction to Snapshot Algorithms in Distributed Computing," *Distributed Systems Eng.,* vol. 2, no. 4, pp. 224-233, 1995.
[12] A. Kshemkalyani and M. Singhal, "Correct Two-Phase and One-Phase Deadlock Detection Algorithms for Distributed Systems," *Proc. IEEE Symp. Parallel and Distributed Processing,* pp. 126-129, 1990.
[13] A. Kshemkalyani and B. Wu, "Nonintrusive Snapshots Using Thin Slices," *Proc. Int'l Conf. Embedded and Ubiquitous Computing,* pp. 572-583, 2005.
[14] T.-H. Lai and T. Yang, "On Distributed Snapshots," *Information Processing Letters,* vol. 25, no. 3, pp. 153-158, 1987.
[15] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, pp. 558-565, 1978.
[16] H.F. Li, T. Radhakrishnan, and K. Venkatesh, "Global State Detection in Non-FIFO Networks," *Proc. IEEE Int'l Conf. Distributed Computing Systems,* pp. 364-370, 1987.
[17] K. Marzullo and L.S. Sabel, "Efficient Detection of a Class of Stable Properties," *Distributed Computing,* vol. 8, no. 2, pp. 81-91, 1994.
[18] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Proc. Workshop Parallel and Distributed Algorithms,* pp. 215-226, 1989.
[19] F. Mattern, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation," *J. Parallel Distributed Computing,* vol. 18, no. 4, pp. 423-434, 1993.
[20] M. Raynal, *Distributed Algorithms and Protocols.* Wiley and Sons, 1988.
[21] A. Schiper and A. Sandoz, "Strong Stable Properties in Distributed Systems," *Distributed Computing,* vol. 8, no. 2, pp. 93-103, 1994.
[22] M. Spezialetti and P. Kearns, "Efficient Distributed Snapshots," *Proc. IEEE Int'l Conf. Distributed Computing Systems,* pp. 382-388, 1986.
[23] N. Sridhar and P. Sivilotti, "Lazy Snapshots," *Proc. Parallel and Distributed Computing Systems,* pp. 96-101, 2002.
[24] S. Venkatesan and T.-Y. Juang, "Efficient Algorithms for Optimistic Crash Recovery," *Distributed Computing,* vol. 8, no. 2, pp. 105-114, 1994.
[25] S. Venkatesan, "Message-Optimal Incremental Snapshots," *Proc. IEEE Int'l Conf. Distributed Computing Systems,* pp. 53-60, 1989.
[26] M. Weiser, "Programmers Use Slices When Debugging," *Comm. ACM,* vol. 25, no. 7, pp. 446-452, 1982.
[27] "Top 500 Supercomputer Sites," http://www.top500.org, 2006

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

Bin Wu received the BS and MS degrees from the Department of Energy Engineering at Zhejiang University, China, in 1992 and 1995, respectively. He received the MS degree in computer science from the University of Illinois at Chicago in 2002. He is now working toward the PhD degree in computer science at the University of Illinois at Chicago. His research interests include distributed computing, peer-to-peer networks, protocols and algorithms, and Web-based healthcare systems.