An introduction to snapshot algorithms in distributed computing

1995 Distrib. Syst. Engng. 2 224

(http://iopscience.iop.org/0967-1846/2/4/005)

Distrib. Syst. Engng 2 (1995) 224–233. Printed in the UK

An introduction to snapshot algorithms in distributed computing

Ajay D Kshemkalyani†, Michel Raynal‡ and Mukesh Singhal§

† IBM Corporation, PO Box 12195, Research Triangle Park, NC 27709, USA
‡ IRISA, campus de Beaulieu, 35042 Rennes-cedex, France
§ Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210, USA

Received 14 November 1994, in final form 26 July 1995

Abstract. Recording on-the-fly global states of distributed executions is an important paradigm when one is interested in analysing, testing, or verifying properties associated with these executions. Since Chandy and Lamport's seminal paper on this topic, this problem is called *the snapshot problem*. Unfortunately, the lack of both a globally shared memory and a global clock in a distributed system, added to the fact that transfer delays in these systems are finite but unpredictable, makes this problem non-trivial.

This paper first discusses issues which have to be addressed to compute distributed snapshots in a consistent way. Then several algorithms which determine on-the-fly such snapshots are presented for several types of networks (according to the properties of their communication channels, namely, FIFO, non-FIFO, and causal delivery).

1. Introduction

A distributed computing system consists of spatially separated processes that do not share a common memory and communicate asynchronously with each other by passing messages over communication channels. Each component of a distributed system has a local state. The state of a process is characterized by the state of its local memory and a history of its activity. The state of a channel is characterized by the set of messages sent along the channel less the messages received along the channel. The global state of a distributed system is a collection of the local states of its components.

Recording the global state of a distributed system is an important paradigm and it finds applications in several aspects of distributed system design. For examples, in detection of stable properties such as deadlocks [15] and termination [18], the global state of the system is examined for certain properties; for failure recovery, a global state of the distributed system (called a checkpoint) is periodically saved and recovery from a processor failure is done by restoring the system to the last saved global state [14]; for debugging distributed software, the system is restored to a consistent global state [7, 8] and the execution resumes from there in a controlled manner. A snapshot-recording method has been used in the distributed debugging facility of Estelle [12, 10], a distributed programming environment. Other applications include monitoring distributed events [25] such as in industrial process control, setting distributed breakpoints [20], protocol specification and verification [4, 9, 13], and discarding obsolete information [21].

Therefore, it is important that there be efficient ways of recording the global state of a distributed system [6]. Unfortunately, there is no shared memory and no global clock in a distributed system and the distributed nature of the local clocks and local memory makes it difficult to record the global state of the system efficiently.

If shared memory were available, an up-to-date state of the entire system would be available to the processes sharing the memory. The absence of shared memory necessitates ways of getting a coherent and complete view of the system based on the local states of individual processes. A meaningful global snapshot can be obtained if the components of the distributed system record their local states at the same time. This would be possible if the local clocks at processes were perfectly synchronized or if there were a global system clock that could be instantaneously read by the processes. However, it is technologically unfeasible to have perfectly synchronized clocks at various sites – clocks are bound to drift. If processes read time from a single common clock (maintained at one process), various indeterminate transmission delays during the read operation will cause the processes to identify various physical instants as the same time. In both cases, the collection of local state observations will be made at different times and may not be meaningful, as illustrated by the following example.

Example: Let S1 and S2 be two distinct sites of a distributed system which maintain bank accounts A and B, respectively. A site refers to a process in this example. Let the communication channels from site S1 to site S2 and from site S2 to site S1 be denoted by C_{12} and C_{21},

0967-1846/95/040224+10$19.50

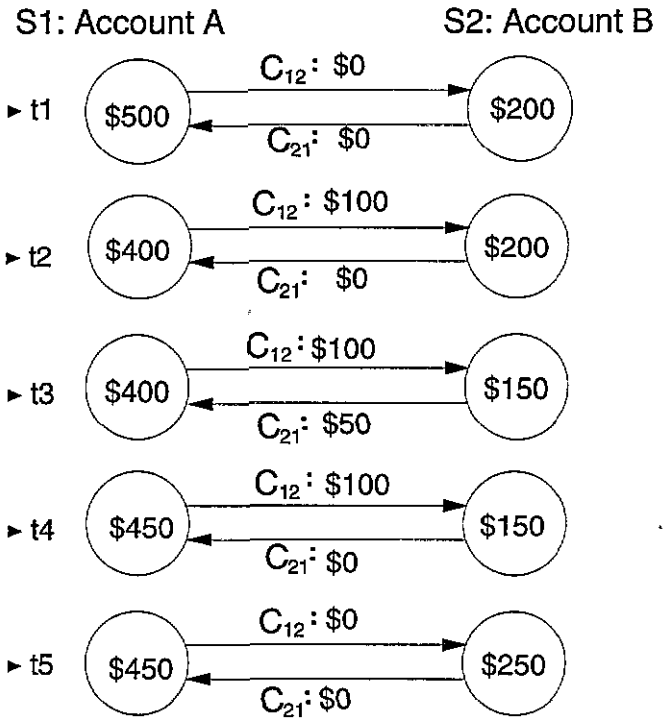

Figure 1. A banking example.

respectively. Consider the following sequence of actions, which are also illustrated in figure 1:

(i) Initially, Account A = \$500, Account B = \$200, C_{12} = \$0, C_{21} = \$0.

(ii) Site S1 initiates a transfer of \$100 from Account A to Account B. Account A is decremented by \$100 to \$400 and a request for \$100 credit to Account B is sent on Channel C_{12} to site S2. Account A = \$400, Account B = \$200, C_{12} = \$100, C_{21} = \$0.

(iii) Site S2 initiates a transfer of \$50 from Account B to Account A. Account B is decremented by \$50 to \$150 and a request for \$50 credit to Account A is sent on Channel C_{21} to site S1. Account A = \$400, Account B = \$150, C_{12} = \$100, C_{21} = \$50.

(iv) Site S1 receives the message for a \$50 credit to Account A and updates Account A. Account A = \$450, Account B = \$150, C_{12} = \$100, C_{21} = \$0.

(v) Site S2 receives the message for a \$100 credit to Account B and updates Account B. Account A = \$450, Account B = \$250, C_{12} = \$0, C_{21} = \$0.

Suppose the local state of Account A is recorded at the end of step 1 to show \$500 and the local state of Account B and channels C_{12} and C_{21} are recorded at the end of step 3 to show \$150, \$100, and \$50, respectively. Then the recorded global state shows \$800 in the system. An extra of \$100 appears in the system. The reason for the inconsistency is that Account A's state was recorded before the \$100 transfer to Account B using channel C_{12} was initiated, whereas channel C_{12}'s state was recorded after the \$100 transfer was initiated.

This simple example shows that recording a consistent global state of a distributed system is not a trivial task. This paper addresses this fundamental issue of distributed computing. The work presented in this paper will be useful to designers of distributed systems and designers of application support mechanisms.

The rest of the paper is organized as follows. Section 2 presents the system model and a formal definition of the notion of consistent global state. The subsequent sections present algorithms to record such global states under various communication models. These algorithms are called snapshot algorithms. Section 3 presents snapshot algorithms for FIFO communication channels. It presents the Chandy–Lamport snapshot algorithm followed by a short discussion on three variations of it. Section 4 presents snapshot algorithms for non-FIFO communication channels. Section 5 discusses algorithms for sytems that support causal ordering of messages. Finally, Section 6 concludes the paper with summary remarks.

2. System model and definitions

2.1. System model

The system consists of a collection of n processes, indexed from 1 to n, that are connected by channels. There is no globally shared memory and processes communicate solely by passing messages. There is no physical global clock in the system. Message send and receive is asynchronous. Messages are delivered reliably with finite but arbitrary time delay. The system can be described as a directed graph in which vertices represent the processes and edges represent unidirectional communication channels. Let C_{ij} denote the channel from process i to process j.

Processes and channels have states associated with them. The state of a process at any time is defined by the contents of processor registers, stacks, local memory, etc and may be highly dependent on the local context of the distributed application. The state of channel C_{ij}, denoted by SC_{ij}, is given by the set of messages in transit in the channel.

The actions performed by a process are modelled as three types of events, namely, internal events, message send events, and message receive events. For a message m_{ij} that is sent by process i to process j, let $send(m_{ij})$ and $rec(m_{ij})$ denote its send and receive events, respectively. Occurrence of events changes the states of respective processes and channels, thus causing transitions in global system state. For example, an internal event changes the state of the process at which it occurs. A send event (or a receive event) changes the state of the process that sends (or receives) the message and the state of the channel on which the message is sent (or received). The events at a process are linearly ordered by their order of occurrence.

At any instant, the state of process i, denoted by LS_i, results from the sequence of all the events executed by process i till that instant. For an event e and a process state LS_i, $e \in LS_i$ iff e belongs to the sequence of events that have taken process i to state LS_i.

A channel is a distributed entity and its state depends on the local states of the processes on which it is incident. For a channel C_{ij}, the following set of messages can be defined, based on the local states of the processes i and j [11].

Transit: $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \notin LS_j\}$

Thus, if a snapshot recording algorithm records the state of processes i and j as LS_i and LS_j, respectively, then it must record the state of channel C_{ij} as $transit(LS_i, LS_j)$.

There are several models of communication among processes and different snapshot algorithms have assumed different models of communication. In a FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel. In non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order. The 'causal ordering' model [5] is based on Lamport's 'happens before' relation on the system events. An event e_1 *happens before* event e_2, denoted by $e_1 \rightarrow e_2$, if (a) e_1 occurs before e_2 on the same process, or (b) e_1 is the send event of a message and e_2 is the receive event of that message, or (c) $\exists e' | e_1$ happens before e' and e' happens before e_2. A system that supports a causal ordering model satisfies the following property:

CO: For any two messages m_{ij} and m_{kj}, if $send(m_{ij}) \longrightarrow send(m_{kj})$, then $rec(m_{ij}) \longrightarrow rec(m_{kj})$.

Causally ordered delivery of messages implies FIFO message delivery. Causal ordering model is useful in developing distributed algorithms and may simplify the algorithms themselves.

2.2. Global state

The global state of a distributed system is a collection of the local states of the processes and the channels. Notationally, a global state GS is defined as

$$GS = \{\bigcup_i LS_i, \bigcup_{i,j} SC_{ij}\}$$

A global state GS is a *consistent global state* iff it satisfies the following two conditions:

C1: send$(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus$ rec$(m_{ij}) \in LS_j$. ($\oplus$ is Ex-OR operator.)

C2: send$(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge$ rec$(m_{ij}) \notin LS_j$.

In a consistent global state, every message that is recorded as received is also recorded as sent and such a state captures the notion of causality that a message cannot be received if it was not sent. Consistent global states are meaningful global states and inconsistent global states are not meaningful in the sense that a distributed system can never be in an inconsistent state.

2.3. Issues in recording a global state

If a global physical clock were available, the following simple procedure could be used to record a consistent global snapshot of a distributed system: The initiator of the snapshot collection decides a future time at which the snapshot is to be taken and broadcasts this time to each process. All processes take their local snapshots at that instant in the global time. The snapshot of channel C_{ij} includes all the messages that process j receives after taking the snapshot and whose timestamp is smaller than the time of the snapshot. (All messages are timestamped with the sender's time.) Clearly, if channels are not FIFO, a termination detection scheme will be needed to determine when to stop waiting for messages on channels.

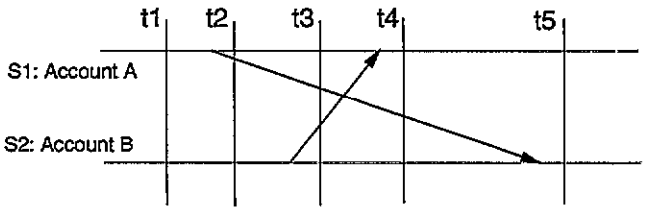

Figure 2. Timing diagram for the banking example.

However, a global physical clock is not available in a distributed system and the following two issues need to be addressed in recording a consistent global snapshot of a distributed system:

I1: How to distinguish between the messages to be recorded in the snapshot (either in a channel state or a process state) from those not to be recorded. The answer to this comes from conditions **C1** and **C2** as follows:
Any message that is sent by a process before recording its snapshot must be recorded in the global snapshot (from **C1**).
Any message that is sent by a process after recording its snapshot must not be recorded in the global snapshot (from **C2**).

I2: How to determine the instant when a process takes its snapshot. The answer to this comes from condition **C2**: A process j must record its snapshot before processing a message m_{ij} that was sent by process i after recording its snapshot.

2.4. Cuts of a distributed computation

A distributed computation can be conveniently represented using a timing diagram where horizontal lines represent the processes' time lines. Figure 2 shows a timing diagram for the computation illustrated in figure 1. A line joining one arbitrary point on each process line slices the timing diagram into a PAST and a FUTURE. Such a line is termed a *cut* in the computation. Every cut corresponds to a global state and every global state can be graphically represented as a cut in the computation's timing diagram [3]. A consistent global state corresponds to a cut in which every message received in the PAST of the cut has been sent in the PAST of that cut. Such a cut is known as a *consistent cut*. Cuts in a timing diagram provide a powerful graphical aid in representing and reasoning about global states of a computation.

We next discuss a set of representative snapshot algorithms for distributed systems. These algorithms assume different interprocess communication capabilities about the underlying system and illustrate how interprocess communication affects the design complexity of these algorithms. There are two types of messages: computation messages and control messages. The former are exchanged

Marker Sending Rule for process i

(i) Process i records its state.
(ii) For each outgoing channel C on which a marker has not been sent, i sends a marker along C before i sends further messages along C.

Marker Receiving Rule for process j

On receiving a marker along channel C:
if j has not recorded its state **then**
begin Record the state of C as the empty set
Follow the 'Marker Sending Rule'
end
else
Record the state of C as the set of messages received along C after j's state was recorded and before j received the marker along C

Figure 3. The Chandy–Lamport algorithm.

by the underlying application and the latter are exchanged by the snapshot algorithm. Execution of a snapshot algorithm is transparent to the underlying application, except for occasional delaying of some actions of the application.

3. Snapshot algorithms for FIFO channels

This section presents Chandy and Lamport algorithm [6], which was the first algorithm to record the global snapshot, and three of its variations.

3.1. Chandy–Lamport algorithm

3.1.1. Principle. After a site has recorded its snapshot, it sends a control message, called a *marker*, along all its outgoing channels before sending out any more messages. Since channels are FIFO, a marker separates the messages in the channel into those to be included in the snapshot (i.e. channel state or process state) from those not to be recorded in the snapshot. (This addresses issue **I1.**) The role of markers in a FIFO system is to act as delimiters for the messages in the channels so that the channel state recorded by the process at the receiving end of the channel satisfies the condition **C2.**

Since all messages that follow a marker on channel C_{ij} have been sent by process i after i has taken its snapshot, process j must record its snapshot not later than when it receives a marker on channel C_{ij}. (This addresses issue **I2.**)

3.1.2. The algorithm. The algorithm is given in figure 3. A process initiates snapshot collection by executing the 'Marker Sending Rule' by which it records its local state and sends a marker on each outgoing channel. A process executes the 'Marker Receiving Rule' on receiving a marker. If the process has not yet recorded its local state, it executes the 'Marker Sending Rule' to record its local state. The state of the incoming channel on which the marker is received is recorded as being the set of computation messages received on that channel after

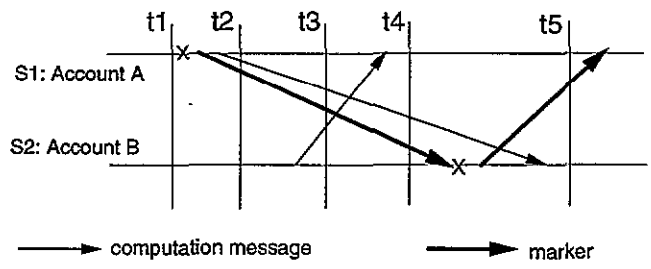

Figure 4. Timing diagram of the snapshot algorithm for the banking example.

recording the local state but before receiving the marker on that channel. The algorithm can be initiated by any process by executing the 'Marker Sending Rule'.

To prove the correctness of the algorithm, we now show that a recorded snapshot satisfies conditions **C1** and **C2.** Since a process records its snapshot when it receives the first marker on any incoming channel, no messages that follow markers on the channels incoming to it are recorded in the process's snapshot. Moreover, a process stops recording the state of an incoming channel when a marker is received on that channel. Due to the FIFO property of channels, it follows that no message sent after the marker on that channel is recorded in the channel state. Thus, condition **C2** is satisfied. When a process j receives message m_{ij} that precedes the marker on channel C_{ij}, it acts as follows: if process j has not taken its snapshot yet, then it includes m_{ij} in its recorded snapshot. Otherwise, it records m_{ij} in the state of the channel C_{ij}. Thus, condition **C1** is satisfied.

The recorded local snapshots can be put together to create the global snapshot in several ways. One policy is to have each process send its local snapshot to the initiator of the algorithm. Another policy is to have each process send the information it records along all outgoing channels, and to have each process receiving such information for the first time propagate it along its outgoing channels. All the local snapshots get disseminated to all other processes and all the processes can determine the global state.

The recording part of a single instance of the algorithm requires $O(e)$ messages and $O(d)$ time, where e is the number of edges in the graph and d is the diameter of the graph.

3.2. Property of the recorded global state

The recorded global state may not correspond to any of the global states that occurred during the computation. Consider a possible execution of the snapshot algorithm for the money transfer example of figure 2 using a timing diagram in figure 4. Let site S1 initiate the algorithm at the end of step 1. Site S1 records its local state (Account A = \$500) and sends a marker to site 2. The marker is received by site S2 at the end of step 4. When site S2 receives the marker, it records its local state (Account B = \$250), the state of channel **C1** as \$0, and sends a marker along channel **C2.** When site S1 receives this marker, it records the state of Channel **C2** as \$50. The \$700 amount in the system is conserved in the recorded global state. However, this global

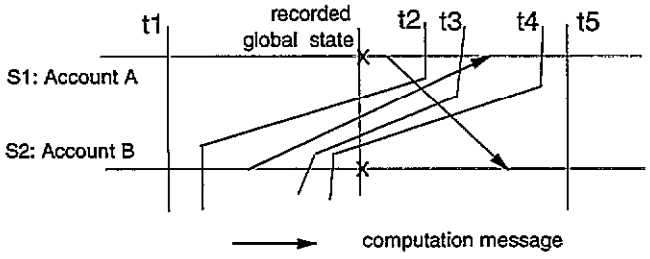

Figure 5. Applying the rubber-band criterion.

state never occurred in the execution. This happens because a process can change its state asynchronously before the markers it sent are received by other sites and the other sites record their states.

Nevertheless, as we discuss next, the system could have passed through the recorded global state in an equivalent execution [6]. Suppose the algorithm is initiated in global state S_i and it terminates in global state S_t. Let *seq* be the sequence of events which takes the system from S_i to S_t. Let S^* be the global state recorded by the algorithm. Chandy and Lamport [6] showed that there exists a sequence *seq'* which is a permutation of *seq* such that S^* is reachable from S_i by executing a prefix of *seq'* and S_t is reachable from S^* by executing the rest of the events of *seq'*.

Thus, the recorded global state is a valid state in an equivalent execution and if a stable property (i.e. a property that persists, such as termination or deadlock) holds in the system before the snapshot algorithm begins, it holds in the recorded global snapshot. Therefore, a recorded global state is useful in detecting stable properties.

A physical interpretation of the collected global state is as follows. Consider the two instants of recording of the local states in the banking example. These instants are marked by crosses in figure 4. If the cut formed by these instants is viewed as being an elastic band and if the elastic band is stretched so that it is vertical, then all the recorded states of all processes occur simultaneously at one physical instant and the recorded global state occurs in the execution that is depicted in this modified timing diagram (figure 5). Note that the system execution would have been like this, had the processors' speeds and message delays been different. Yet another physical interpretation of the collected global state is as follows: all the recorded process states are mutually concurrent—no process state causally depends upon another. Therefore, we can view logically that all these process states occurred simultaneously even though they might have occurred at different instants in physical time.

3.3. Variations of the Chandy–Lamport algorithm

Several variants of the Chandy–Lamport snapshot algorithm followed. These variants refined and optimized the basic algorithm. For example, Spezialetti and Kearns algorithm [24] optimizes concurrent initiation of snapshot collection and efficiently distributes the recorded snapshot. Venkatesan's algorithm [23] optimizes the basic snapshot algorithm to efficiently record repeated snapshots of a distributed system that are required in recovery algorithms with synchronous checkpointing.

Spezialetti–Kearns method: There are two phases in obtaining a global snapshot: locally recording the snapshot at every process and distributing the resultant global snapshot to all the initiators. Spezialetti and Kearns [24] optimized the Chandy–Lamport algorithm by exploiting the work of combining concurrently initiated snapshots (in the first phase) to efficiently distribute the resultant global snapshot to only the concurrent initiators (in the second phase). A process needs to take only one snapshot, irrespective of the number of concurrent initiators and all processes are not sent the global snapshot.

This algorithm assumes bidirectional channels in the system. The message complexity of snapshot recording is $O(e)$ irrespective of the number of concurrent initiations of the algorithm. The message complexity of assembling and disseminating the snapshot is $O(rn^2)$ where r is the number of concurrent initiations.

Venkatesan's incremental snapshot method: Many applications require repeated collection of global snapshots of the system. For example, recovery algorithms with synchronous checkpointing need to advance their checkpoints periodically. This can be achieved by repeated invocations of the Chandy–Lamport algorithm. However, Venkatesan [23] proposed the following efficient approach: Execute an algorithm to record an incremental snapshot since the most recent snapshot was taken and combine it with the most recent snapshot to obtain the latest snapshot of the system. The incremental snapshot algorithm of Venkatesan [23] modifies the global snapshot algorithm of Chandy–Lamport to save on messages when computation messages are sent only on a few of the network channels, between the recording of two successive snapshots.

The incremental snapshot algorithm assumes bidirectional FIFO channels, the presence of a single initiator, a fixed spanning tree in the network, and four types of control messages: *init_snap*, *snap_completed*, *regular*, and *ack*. *init_snap* and *snap_completed* messages traverse spanning edges. *regular* and *ack* messages which serve to record states of non-spanning edges are not sent on those edges on which no computation message has been sent since the previous snapshot.

Venkatesan [23] showed that the lower bound on the message complexity of an incremental snapshot algorithm is $\Omega(u + n)$ where u is the number of edges on which a computation message has been sent since the previous snapshot. Venkatesan's algorithm achieves this lower bound in message complexity.

Helary's wave synchronization method: Helary's snapshot algorithm [11] incorporates the concept of message waves in the Chandy–Lamport algorithm. A wave is a flow of control messages such that every process in the system is visited exactly once by a wave control message, and at least one process in the system can determine when this flow of control messages terminates. A wave is initiated after the previous wave terminates. Wave sequences may be implemented by various traversal structures such as a ring. A process begins recording

the local snapshot when it is visited by the wave control message.

Note that in this algorithm, the primary function of wave synchronization is to evaluate functions over the recorded global snapshot. This algorithm has a message complexity of $O(e)$ to record a snapshot (because all channels can be traversed to implement the wave).

4. Snapshot algorithms for non-FIFO channels

A FIFO system ensures that all messages sent after a marker on a channel will be delivered after the marker. This ensures that condition **C2** is satisfied in the recorded snapshot if LS_i, LS_j, and SC_{ij} are recorded as described in the Chandy–Lamport algorithm. In a non-FIFO system, the problem of global snapshot recording is complicated because a marker cannot be used to delineate messages into those to be recorded in the global state from those not to be recorded in the global state. In such systems, different techniques have to be used to ensure that a recorded global state satisfies condition **C2**.

In a non-FIFO system, either some degree of inhibition (i.e. temporarily delaying the execution of an application process or delaying the send of a computation message) or piggybacking of control information on computation messages to capture out-of-sequence messages, is necessary to record a consistent global snapshot [22]. The non-FIFO algorithm by Helary uses message inhibition.[11]. The non-FIFO algorithms by Lai and Yang [16], Li *et al* [17] and Mattern [19] use message piggybacking to distinguish computation messages sent after the marker from those sent before the marker.

The non-FIFO algorithm of Helary [11] uses message inhibition to avoid an inconsistency in a global snapshot in the following way: When a process receives a marker, it immediately returns an acknowledgement. After a process i has sent a marker on the outgoing channel to process j, it does not send any messages on this channel until it is sure that j has recorded its local state. Process i can conclude this if it has received an acknowledgement for the marker sent to j, or has received a marker for this snapshot from j.

We next discuss snapshot recording algorithms for systems with non-FIFO channels that use piggybacking of computation messages.

4.1. Lai–Yang algorithm

Lai and Yang's global snapshot algorithm for non-FIFO systems [16] is based on two observations on the role of a marker in a FIFO system. The first observation is that a marker ensures that condition **C2** is satisfied for LS_i and LS_j when the snapshots are recorded at processes i and j, respectively. The Lai–Yang algorithm fulfills this role of a marker in a non-FIFO system by using a colouring scheme on computation messages as follows.

(i) Every process is initially white and turns red while taking a snapshot. The equivalent of the 'Marker Sending Rule' is executed when a process turns red.

(ii) Every message sent by a white (red) process is coloured white (red). Thus, a white (red) message is a message that was sent before (after) the sender of that message recorded its local snapshot.

(iii) Every white process takes its snapshot at its convenience, but no later than the instant it receives a red message.

Thus, when a white process receives a red message, it records its local snapshot before processing the message. This ensures that no message sent by a process after recording its local snapshot is processed by the destination process before the destination records its local snapshot. Thus, an explicit marker message is not required in this algorithm and the 'marker' is piggybacked on computation messages using a colouring scheme.

The second observation is that the marker informs process j of the value of $\{send(m_{ij}) | send(m_{ij}) \in LS_i\}$ so that $transit(LS_i, LS_j)$ can be computed. The Lai–Yang algorithm fulfils this role of the marker in the following way.

(iv) Every white process records a history of all white messages sent or received by it along each channel.

(v) When a process turns red, it sends these histories along with its snapshot to the initiator process that collects the global snapshot.

(vi) The initiator process evaluates $transit(LS_i, LS_j)$ for each channel C_{ij} as given below:
$SC_{ij} = \{send(m_{ij}) | send(m_{ij}) \in LS_i\} - \{rec(m_{ij}) | rec(m_{ij}) \in LS_j\}$.

Condition **C2** holds because a red message is not included in the snapshot of the recipient process and a channel state is the difference of two sets of white messages. Condition **C1** holds because a white message m_{ij} is included in the snapshot of process j if j receives m_{ij} before taking its snapshot. Otherwise, m_{ij} is included in the state of channel C_{ij}.

Though marker messages are not required in the algorithm, each process has to record the entire message history on each channel as part of the local snapshot. Thus, the space requirements of the algorithm may be large. Lai and Yang describe how the size of the local storage and snapshot recording can be reduced by storing only the messages sent and received since the previous snapshot recording, assuming that the previous snapshot is still available. This approach can be very useful to applications that require repeated snapshots of a distributed system.

4.2. Li *et al*'s algorithm

Li *et al*'s algorithm [17] for recording a global snapshot in a non-FIFO system is similar to the Lai–Yang algorithm. Markers are tagged so as to generalize the red/white colours of the Lai–Yang algorithm to accommodate repeated invocations of the algorithm and multiple initiators. In addition, the algorithm is not concerned with the contents of computation messages and the state of a channel is computed as the number of messages in transit in the channel. This simplification is combined with the

incremental technique to compute channel states, also outlined by Lai and Yang, which reduces the size of message histories to be stored and transmitted. The initiator computes the state of C_{ij} as: (the number of messages in C_{ij} in the previous snapshot) + (the number of messages sent on C_{ij} since the last snapshot at i) − (the number of messages received on C_{ij} since the last snapshot at j).

Though this algorithm does not require any additional message to record a global snapshot provided computation messages are eventually sent on each channel, the local storage and size of tags on computation messages is of size $O(n)$, where n is the number of initiators.

4.3. Mattern's algorithm

Mattern's algorithm [19] is based on vector clocks. In vector clocks, the clock at a process is an integer vector of length n, with one component for each process. The component of a process in the vector clock at a process advances independently whenever the process learns, through messages, that a component value has advanced.

Mattern's algorithm assumes a single initiator process and works as follows.

(i) The initiator 'ticks' its local clock and selects a future vector time s at which it would like a global snapshot to be recorded. It then broadcasts this time s and freezes all activity until it receives acknowledgements of the receipt of this broadcast.

(ii) When a process receives the broadcast, it remembers the value s and returns an acknowledgement to the initiator.

(iii) After having received an acknowledgement from every process, the initiator increases its vector clock to s and broadcasts a dummy message to all processes. (Observe that before broadcasting this dummy message, the local clocks of other processes have a value $\not\geq s$.)

(iv) The receipt of this dummy message forces each recipient to increase its clock to a value $\geq s$ if not already $\geq s$.

(v) Each process takes a local snapshot and sends it to the initiator when (just before) its clock increases from a value less than s to a value $\geq s$. Observe that this may happen before the dummy message arrives at the process.

(vi) The state of C_{ij} is all messages sent along C_{ij}, whose timestamp is smaller than s and which are received by p_j after recording LS_j.

Processes record their local snapshot as per rule (5). Any message m_{ij} sent by process i after it records its local snapshot LS_i has a timestamp $> s$. Assume that this m_{ij} is received by j before it records LS_j. After receiving this m_{ij} and before j records LS_j, j's local clock reads a value $> s$, as per rules for updating vector clocks. This implies j must have already recorded LS_j as per rule (5), which contradicts the assumption. Therefore, m_{ij} cannot be received by j before it records LS_j. By rule (6), m_{ij} is not recorded in SC_{ij} and therefore, condition **C2** is satisfied. Condition **C1** holds because each message m_{ij} with a timestamp less than s is included in the snapshot of process j if j receives m_{ij} before taking its snapshot. Otherwise, m_{ij} is included in the state of channel C_{ij}.

The following observations about the above algorithm lead to various optimizations. (i) The initiator can be made a 'virtual' process; so, no process has to freeze. (ii) As long as a new higher value of s is selected, the phase of broadcasting s and returning the acks can be eliminated. (iii) Only the initiator's component of s is used to determine when to record a snapshot. Also, one needs to know only if the initiator's component of the vector timestamp in a message has increased beyond the value of the corresponding component in s. Therefore, it suffices to have just two values of s, say, white and red, which can be represented using one bit.

With these optimizations, the algorithm becomes similar to the Lai–Yang algorithm except for the manner in which $transit(LS_i, LS_j)$ is evaluated for channel C_{ij}. In Mattern's algorithm, a process is not required to store message histories to evaluate the channel states. The state of any channel is the set of all the white messages that are received by a red process on which that channel is incident. A termination detection scheme for non-FIFO channels is required to detect that no white messages are in transit to ensure that the recording of all the channel states is complete.

The savings of not storing and transmitting entire message histories, over the Lai–Yang algorithm, comes at the expense of delay in the termination of the snapshot recording algorithm and need for a termination detection scheme (e.g. a message counter per channel).

5. Snapshots in a causal delivery system

Two global snapshot-recording algorithms, namely, Acharya–Badrinath [1] and Alagar–Venkatesan [2] assume that the underlying system supports causal message delivery. The causal message delivery property **CO** provides a built-in message synchronization to control and computation messages. Consequently, snapshot algorithms for such systems are considerably simplified. For example, these algorithms do not send control messages (i.e. markers) on every channel and are simpler than the snapshot algorithms for a FIFO system.

Both these algorithms use an identical principle to record the state of processes. An initiator process broadcasts a token, denoted as *token*, to every process including itself. Let the copy of the token received by process i be denoted $token_i$. A process i records its local snapshot LS_i when it receives $token_i$ and sends the recorded snapshot to the initiator.

These algorithms do not require each process to send markers on each channel, and the processes do not coordinate their local snapshot recordings with every other process. Nonetheless, for any two processes i and j the following property (called Property **P1**) is satisfied:

$$send(m_{ij}) \notin LS_i \Rightarrow rec(m_{ij}) \notin LS_j.$$

This is due to the causal ordering property of the underlying system as explained next. Let a message

Table 1. Comparison of the snapshot algorithms.

Algorithms	Features
Chandy–Lamport [6], 1985	Baseline algorithm. FIFO systems. $O(e)$ messages to record snapshot.
Spezialetti–Kearns [24], 1986	Improvements to [6]: supports concurrent initiators, efficient assembly and distribution of snapshot. Assumes bidirectional channels. $O(e)$ messages to record, $O(rn^2)$ messages to assemble and distribute snapshot.
Venkatesan [23], 1989	Based on [6]. Selective sending of markers. Provides message-optimal incremental snapshots. $\Omega(n+u)$ messages to record snapshot.
Helary [11], 1989	Based on [6]. Uses wave synchronization. Evaluates function over recorded global state. Adaptable to non-FIFO systems but requires inhibition.
Lai–Yang [16], 1987	Non-FIFO system. Markers piggybacked on computation messages. Message history required to compute channel states.
Li *et al* [17], 1987	Similar to [16]. Small message history needed as channel states are computed incrementally.
Mattern [19], 1989	Similar to [16]. No message history required. Termination detection (e.g. a message counter per channel) required to compute channel states.
Acharya–Badrinath [1], 1992	Requires causal delivery support, Centralized computation of channel states, Channel message contents not known. Requires $2n$ messages, 2 time units.
Alagar–Venkatesan [2], 1993	Requires causal delivery support, Distributed computation of channel states. Requires $3n$ messages, 3 time units, small messages.

n = # processes, u = # edges on which messages were sent after previous snapshot,
e = # channels, r = # concurrent initiators.

m_{ij} be such that $rec(token_i) \longrightarrow send(m_{ij})$. Then $send(token_j) \longrightarrow send(m_{ij})$ and the underlying causal ordering property ensures that $rec(token_j)$, at which instant j records LS_j, happens before $rec(m_{ij})$. Thus, m_{ij} whose send is not recorded in LS_i, is not recorded as received in LS_j.

Methods of channel state recording are different in these two algorithms and are discussed next.

5.1. Channel Recording in the Acharya–Badrinath algorithm

Each process i maintains arrays $SENT_i[1,\ldots,N]$ and $RECD_i[1,\ldots,N]$. $SENT_i[j]$ is the number of messages sent by process i to process j and $RECD_i[j]$ is the number of messages received by process i from process j. The arrays may not contribute to the storage complexity of the algorithm because the underlying causal ordering protocol may require these arrays to enforce causal ordering.

Channel states are recorded as follows: when a process i records its local snapshot LS_i on the receipt of $token_i$, it includes arrays $RECD_i$ and $SENT_i$ in its local state before sending the snapshot to the initiator. When the algorithm terminates, the initiator determines the state of channels in the global snapshot being assembled as follows:

(i) The state of each channel from the initiator to each process is empty.
(ii) The state of channel from process i to process j is the set of messages whose sequence numbers are given by $\{RECD_j[i]+1,\ldots,SENT_i[j]\}$.

We now show that the algorithm satisfies conditions **C1** and **C2**.

Let a message m_{ij} be such that $rec(token_i) \longrightarrow send(m_{ij})$. Clearly, $send(token_j) \longrightarrow send(m_{ij})$ and the sequence number of m_{ij} is greater than $SENT_i[j]$. Therefore, m_{ij} is not recorded in SC_{ij}. Thus, send$(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij}$. This in conjunction with property **P1** implies that the algorithm satisfies condition **C2**.

Consider a message m_{ij} which is the kth message from process i to process j before i takes its snapshot. The two possibilities below imply that condition **C1** is satisfied.

- Process j receives m_{ij} before taking its snapshot. In this case, m_{ij} is recorded in j's snapshot.
- Otherwise, $RECD_j[i] \leq k \leq SENT_i[j]$ and the message m_{ij} will be included in the state of channel C_{ij}.

This algorithm requires $2n$ messages and 2 time units for recording and assembling the snapshot, where one time unit is required for the delivery of a message. If the contents of messages in channel states are required, the algorithm requires $2n$ messages and 2 time units additionally.

5.2. Channel recording in the Alagar–Venkatesan algorithm

A message is referred to as *old* if the send of the message causally precedes the send of the token. Otherwise, the message is referred to as *new*. Whether a message is new or old can be determined by examining the vector timestamp in the message, which is needed to enforce causal ordering among messages.

In the Alagar–Venkatesan algorithm [2], channel states are recorded as follows.

(i) When a process receives the *token*, it takes its snapshot, initializes the state of all channels to empty, and returns a *Done* message to the initiator. Now onwards, a process includes a message received on a channel in the channel state only if it is an old message.
(ii) After the initiator has received a *Done* message from all processes, it broadcasts a $Terminate$ message.
(iii) A process stops the snapshot algorithm after receiving a $Terminate$ message.

An interesting observation is that a process receives all the old messages in its incoming channels before it receives the *Terminate* message. This is ensured by the underlying causal message delivery property.

Causal ordering property ensures that no new message is delivered to a process prior to the *token* and only old messages are recorded in the channel states. Thus, $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij}$. This together with Property **P1** implies that condition **C2** is satisfied. Condition **C1** is satisfied because each old message m_{ij} is delivered either before the token is delivered or before the *Terminate* is delivered to a process and thus gets recorded in LS_i or SC_{ij}, respectively.

6. Summary

Recording global state of a distributed system is an important paradigm in the design of the distributed systems and the design of efficient methods of recording the global state is an important issue. Recording of a global state of a distributed system is complicated due to the lack of both a globally shared memory and a global clock in a distributed system. This paper first presented a formal definition of the global state of a distributed system and exposed issues related to its capture; it then described several algorithms to record a snapshot of a distributed system under various communication models.

Table 1 gives a comparison of the salient features of the various snapshot-recording algorithms. Clearly, the higher the level of abstraction provided by a communication model, the simpler the snapshot algorithm. However, there is no best-performing snapshot algorithm and an appropriate algorithm can be chosen based on the application's requirement. For examples, for termination detection, a snapshot algorithm that computes a channel state as the number of messages is adequate; for checkpointing for recovery from failures, an incremental snapshot algorithm is likely to be the most efficient; for global state monitoring, rather than recording and evaluating complete snapshots at regular intervals, it is more efficient to monitor changes to the variables that affect the predicate and evaluate the predicate only when some component variable changes.

As indicated in the introduction, the paradigm of global snapshots finds a large number of applications (among others: detection of stable properties, checkpointing, monitoring, debugging, analyses of distributed computation, discarding of obsolete information). Moreover, in addition to the problems they solve, the algorithms presented in this paper are of great importance to people interested in distributed computing, since these algorithms illustrate the incidence of properties of communication channels (FIFO, non-FIFO, causal ordering) on the design of a class of distributed algorithms.

Acknowledgments

The authors are grateful to Professors F Mattern and S Venkatesan for providing useful feedback on an earlier version of the paper.

References

[1] Acharya A and Badrinath B R 1992 Recording distributed snapshots based on causal order of message delivery *Information Processing Lett.* **44** 317–21

[2] Alagar S and Venkatesan S 1994 An optimal algorithm for distributed snapshots with causal message ordering *Information Processing Lett.* **50** 311–6

[3] Babaoglu O and Marzullo K 1993 Consistent global states of distributed systems: fundamental concepts and mechanisms *Distributed Systems* ed S J Mullender (ACM Press) ch 4

[4] Babaoglu O and Raynal M 1995 Specification and verification of dynamic properties in distributed computations *J. Parallel Distributed Systems* **28**

[5] Birman K and Joseph T 1987 Reliable communication in presence of failures *ACM Trans. Comput. Systems* **3** 47–76

[6] Chandy K M and Lamport L 1985 Distributed snapshots: determining global states of distributed systems *ACM Trans. Comput. Systems* **3** 63–75

[7] Cooper R and Marzullo K 1991 Consistent detection of global predicates *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging (May 1991)* pp 163–73

[8] Fromentin E, Plouzeau N and Raynal M 1995 An introduction to the analysis and debug of distributed computations *Proc. 1st IEEE Int. Conf. on Algorithms and Architectures for Parallel Processing (Brisbane, April 1995)* pp 545–54

[9] Geihs K and Seifert M 1986 Automated validation of a cooperation protocol for distributed systems *Proc. 6th Int. Conf. on Distributed Computing Systems* pp 436-43

[10] Gerstel O, Hurfin M, Plouzeau N, Raynal M and Zaks S 1995 On-the-fly replay: a practical paradigm and its implementation for distributed debugging *Proc. 6th IEEE Int. Symp. on Parallel and Distributed Debugging (Dallas, TX, Oct. 1995)* pp 266–72

[11] Helary J-M 1989 Observing global states of asynchronous distributed applications *Proc. 3rd Int. Workshop on Distributed Algorithms, LNCS 392* (Berlin: Springer) pp 124–34

[12] Hurfin M, Plouzeau N and Raynal M 1993 A debugging tool for distribted Estelle programs *J. Comput. Commun.* **16** 328–33

[13] Kamal J and Singhal M 1992 Specification and verification of distributed mutual exclusion algorithms *Technical Report* (Columbus, OH: The Ohio State University, Department of Computer and Information Science)

[14] Koo R and Toueg S 1987 Checkpointing and rollback–recovery in distributed systems *IEEE Trans. Software Engineering*

[15] Kshemkalyani A and Singhal M 1994 Efficient detection and resolution of generalized distributed deadlocks *IEEE Trans. Software Engineering* **20** 43–54

[16] Lai T H and Yang T H 1987 On distributed snapshots *Information Processing Lett.* **25** 153–8

[17] Li H F, Radhakrishnan T and Venkatesh K 1987 Global state detection in non-FIFO networks *Proc. 7th Int. Conf. on Distributed Computing Systems* pp 364–70

[18] Mattern F 1987 Algorithms for distributed termination detection *Distributed Computing* pp 161–75

[19] Mattern F 1993 Efficient algorithms for distributed snapshots and global virtual time approximation *J. Parallel Distributed Computing* **18** 423–34

[20] Miller B and Choi J 1988 Breakpoints and halting in distributed programs *Proc. 8th Int. Conf. on Distributed Computing Systems* pp 316–23

[21] Sarin S and Lynch N 1987 Discarding obsolete information in a replicated database system *IEEE Trans. Software Engineering* **13** 39–47

[22] Taylor K 1989 The role of inhibition in consistent cut protocols *Proc. 3rd Int. Workshop on Distributed Algorithms LNCS 392* (Berlin: Springer) pp 124–34
[23] Venkatesan S 1993 Message-optimal incremental snapshots *J. Comput. Software Engineering* **1** 211–31
[24] Spezialetti M and Kearns P 1986 Efficient distributed snapshots *Proc. 6th Int. Conf. on Distributed Computing Systems* pp 382–8
[25] Spezialetti M and Kearns P 1989 Simultaneous regions: a framework for the consistent monitoring of distributed systems *Proc. 9th Int. Conf. on Distributed Computing Systems* pp 61–8