[8] S. Sarin, N. Lynch, *Discarding Obsolete Information in a Replicated Database System,* IEEE Transactions on Software Engineering, 13(1), 39-47, 1987.

[9] K. Taylor, *The Role of Inhibition in Consistent Cut Protocols,* Proceedings of the 3rd International Workshop on Distributed Algorithms, LNCS 392, Springer Verlag, 124-134, 1989.

[10] S. Venkatesan, *Message-Optimal Incremental Snapshots,* Proceedings of the 9th International Conference on Distributed Computing Systems, 53-60, 1989.

[11] M. Spezialetti, P. Kearns, *Efficient Distributed Snapshots,* Proceedings of the 6th International Conference on Distributed Computing Systems, 382-388, 1986.

[12] M. Spezialetti, P. Kearns, *Simultaneous Regions: A Framework for the Consistent Monitoring of Distributed Systems,* Proceedings of the 9th International Conference on Distributed Computing Systems, 61-68, 1989.

consistent monitoring of distributed systems to detect global predicates. A process whose local variable is a component of the global predicate informs a monitor whenever the value of the variable changes. This process also coerces other processes to inform the monitor of the values of their variables that are components of the global predicate. The monitor evaluates the global predicate when it recieves the next message from each of the involved processes, informing it of the value(s) of their local variable(s). The periods of local computation on each process between the i^{th} and the $i + 1^{th}$ events at which the values of the local component(s) of the global predicate are reported to the monitor are defined to be the $i + 1^{th}$ simultaneous regions. The above scheme is extended to arrange multiple monitors hierarchically to evaluate complex global predicates.

7 Conclusions

Table 1 gives a comparison of the salient features of the various snapshot recording algorithms. An appropriate algorithm can be chosen based on the application's requirement. Though the technique of recording snapshots is fairly well-understood, they need to be adapted to detect specific global predicates. For example, the detection of global (not necessarliy stable) predicates in a system is an area where snapshot techniques need to be adapted suitably. Schemes to monitor global predicates should be able to monitor the ocurrence of transient states that may not necessarily occur in every run of the computation, states that always occur in every run of the computation, and stable states.

References

[1] A. Acharya, B. R. Badrinath, *Recording Distributed Snapshots Based on Causal Order of Message Delivery,* Information Processing Letters, North Holland, 44, 317-321, 1992.

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

[3] K. Geihs, M. Seifert, *Automated Validation of a Cooperation Protocol for Distributed Systems,* Proceedings of the 6th International Conference on Distributed Computing Systems, 436-443, 1986.

[4] J.-M., Helary, *Observing Global States of Asynchronous Distributed Applications,* Proceedings of the 3rd International Workshop on Distributed Algorithms, LNCS 392, Springer Verlag, 124-134, 1989.

[5] T. H. Lai, T. H. Yang, *On Distributed Snapshots,* Information Processing Letters, North Holland, 25, 153-158, 1987.

[6] H. F. Li, T. Radhakrishnan, K. Venkatesan, *Global State Detection in Non-FIFO Networks,* Proceedings of the 7th International Conference on Distributed Computing Systems, 364-370, 1987.

[7] B. Miller, J. Choi, *Breakpoints and Halting in Distributed Programs,* Proceedings of the 8th International Conference on Distributed Computing Systems, 316-323, 1988.

Algorithms	Features
Chandy-Lamport [2], 1985	Baseline algorithm. FIFO systems. $O(e)$ messages to record snapshot.
Spezialetti-Kearns [11], 1986	Improvements to [2]: supports concurrent initiators, efficient assembly and distribution of snapshot. Assumes bidirectional channels. $O(e)$ messages to record, $O(rn^2)$ messages to asemble and distribute snapshot.
Venkatesan [10], 1989	Based on [2]. Selective sending of markers. Provides message-optimal incremental snapshots. $\Omega(n+u)$ messages to record snapshot.
Helary [4], 1989	Based on [2]. Uses wave synchronization. Evaluates function over recorded global state. Adaptable to non-FIFO systems but requires inhibition.
Acharya-Badrinath [1], 1992	Requires causal delivery support. Requires 2n messages, 2 time units.
Lai-Yang [5], 1987	NonFIFO system. Markers piggybacked on computation messages. Message history required to compute channel states.
Li et al. [6], 1987	Similar to [5]. Small message history needed as channel states are computed incrementally.
Mattern [3], 1989	Similar to [5]. No message history required. Termination detection algorithm required to compute channel states.

Table 1: Comparison of snapshot algorithms.

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

5. 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.

In the above algorithm, $inconsistent(LS_i, LS_j) = \emptyset$, as shown by the following reasoning. 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 $\geq s$. 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 $\geq s$, as per Rules (R1) and (R2) 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. Hence, $inconsistent(LS_i, LS_j) = \emptyset$.

The following observations about the above algorithm can lead to optimizations:
(a) The initiator can be made a "virtual" process so no process has to freeze.
(b) As long as a new higher value of "s" is selected, the phase of broadcasting "s" and returning the acks can be eliminated.
(c) Only the initiator's component of s is used to determine when to record a snapshot. It suffices to have just two values of s, say, white and red, which can be represented using one bit.

The optimized algorithm becomes similar to the Lai-Yang algorithm except for the manner in which $transit(LS_i, LS_j)$ is evaluated. It is not required to store the complete messsage history 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 following scheme can be used for termination detection:
Each red message sent by a process carries a piggybacked value of the number of white messages sent before the local state recording. Each process keeps a counter for the number of white messages received on each channel. A process can detect termination of recording the states of incoming channels when it receives as many white messages on each channel as the value piggybacked on red messages received on that channel.

The savings of not storing and transmitting entire message histories, over the Lai-Yang algorithm, comes about at the expense of delay in the termination of the snapshot recording algorithm and the cost of an explicit termination detection algorithm.

6 Monitoring Global States

Several applications such as debugging need to detect a system state which is determined by the values of variables on a subset of processes. This state can be expressed as a predicate on variables distributed across the involved processes. Rather than recording and evaluating 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.

Spezialetti and Kearns [12] proposed a technique, named simulataneous regions, for the

$C_i[j], j \neq i$, is p_i's best estimate of the logical time of p_j and indicates the time of occurance of the last event on p_j that 'happened before' (See Sidebar 1) the current point in time at p_i. Vector clocks operate as follows:

R1 When an internal event, a message send event or a message receive event occurs on p_i,

$$C_i[i] := C_i[i] + d \qquad (d \geq 0)$$

At a message send event, the message is timestamped with the updated value of $C_i[i]$.

R2 When p_i receives a message m with t_m, update $C_i[i]$ as follows:

$$\forall k, C_i[k] := max(C_i[k], t_m[k])$$

For two clock values C_i and C_j, the following two relations can be defined:
$C_i \leq C_j$ iff $\forall k, C_i[k] \leq C_j[k]$
$C_i \not\leq C_j$ iff $\exists k, C_i[k] \geq C_j[k]$

References

[1] C. J. Fidge, *Timestamps in Message-Passing Systems that Preserve Partial Ordering,* Proc. of the 11th Australian Computer Science Conference, 56-66, 10(1), 1988.

[2] C. J. Fidge, *Logical Time in Distributed Computing Systems,* 28-33, IEEE Computer, August 1991.

[3] F. Mattern, *Virtual Time and Global States of Distributed Systems,* Parallel and Distributed Algorithms, 215-226, M. Cosnard et al. (editors), North Holland, 1989.

End SIDEBAR 2

The algorithm assumes a single initiator process and is described next:

1. The initiator "ticks" its local clock and selects a future vector time s at which it will initiate the snapshot. It then broadcasts this time s and freezes all activity until it receives acknowledgements of the receipt of this broadcast.

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

3. The initiator then 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 $\not\geq s$.)

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

counters for the number of messages sent/received on each incident channel after the previous snapshot. A process records a local snapshot no later than the arrival of a message or a marker with a higher sequence number than that known for previous initiations. When a process records a local snapshot, (i) it records the values of the Input/Output counters and the local state as part of the snapshot. (ii) it updates its record of the sequence number of the last snapshot, and resets the Input/Output counters. (iii) sends the tuple $< init_id, MKNO >$ implicitly or explicitly using markers on all outgoing channels, as described in the previous paragraph. (iv) forwards the recorded snapshot to the initiator.

The algorithm does not distinguish between computation messages and so the computation of the state of a channel in a global snapshot is simplified to computing the number of messages in transit in the channel. This simplification is combined with an incremental technique for computing in-transit messages, also suggested independently in the Lai-Yang algorithm [5], for reducing the size of the entire message history to be locally stored and to be recorded in a local snapshot to compute channel states. The initiator of an algorithm maintains a variable $TRANSIT_{ij}$ for the number of messages in transit in the channel from i to j, as recorded in the previous snapshot. Using the values of Input/Output counters recorded in the new snapshot, the initiator computes $transit(LS_i, LS_j) = TRANSIT_{ij}$ + (# messages sent on that channel since the last snapshot) − (# messages received on that channel since the last snapshot).

If the initiator initiates a snapshot before the completion of the previous snapshot, it is possible that some process may get a message with a lower sequence number after participating in a snapshot initiated later. In this case, the algorithm uses the snapshot with the higher sequence number to also create the snapshot for the lower sequence number. The algorithm works for multiple initiators if separate Input/Output counters are associated with each initiator, and marker messages and fields carry a vector of tuples, one for each initiator.

Though the message complexity of this algorithm is zero, provided that 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. The technique of combining concurrently initiated snapshots as proposed by Spezialetti and Kearns [11] can be combined with this algorithm.

5.3 Mattern's Algorithm

Mattern's algorithm is based on vector clocks. (See Sidebar 2).

Begin SIDEBAR 2

The system of vector clocks was independently formalized by Fidge [1] and Mattern [3]. Vector clocks have been used previously in various contexts such as file consistency, distributed debugging, distributed mutual exclusion, distributed recovery, implementing causal distributed shared memory, execution tracing for testing distributed software, and causal broadcast. Fidge [2] gives an excellent survey of the applications of vector clocks.

Each process p_i is equipped with a clock C_i which is an integer vector of length n. The clock C_i can be thought of as a function that assigns a vector $C_i(a)$ to any event a. $C_i(a)$ is referred to as the timestamp of event a at p_i. $C_i[i]$ corresponds to p_i's own logical time.

5.1 Lai-Yang Algorithm

Lai and Yang's global snapshot algorithm for non-FIFO systems [5] is based on two observations on the role of the marker in a FIFO system. The first observation is that the marker ensures that $inconsistent(LS_i, LS_j) = \emptyset$ for LS_i and LS_j when the snapshots are recorded at processes i and j, respectively. The Lai-Yang algorithm fulfills the role of the marker in a non-FIFO system by piggybacking the 'marker' on every message sent after the local snapshot is recorded. The marker message itself is not required and the 'marker' is piggybacked on computation messages using a coloring scheme. When a process that has not recorded its local snapshot receives a computation message with a piggybacked 'marker', it first records its local snapshot before processing the message. This ensures that no message sent after the local snapshot recording reaches (is processed by) the destination before the destination records its local snapshot. 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 fulfills this role of the marker without using the marker by having some one process assemble the global state and evaluate $transit(LS_i, LS_j)$ for each LS_i and LS_j as $\{send(m_{ij}) | send(m_{ij}) \in LS_i \} - \{rec(m_{ij}) | rec(m_{ij}) \in LS_j \}$. This requires each process to record the entire message history on each channel as part of the local snapshot. By fulfilling the roles of a marker as described above, marker messages are eliminated from the algorithm.

The coloring scheme used to piggyback 'markers' on computation messages is as follows: A white (red) process is a process that has not (has) recorded its local snapshot. A white (red) message is a message that was sent before (after) the sender of that message recorded its local snapshot. 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. Though no markers are required in this algorithm, the space requirements of the algorithm are large because the local snapshot of a process contains the entire message history of that process. 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.

5.2 Li et. al Algorithm

The Li et al. algorithm [6] for recording a global snapshot in a non-FIFO system is similar to the Lai-Yang algorithm. The algorithm as described below is for a single initiator: Snapshots initiated by the initiator are assigned a sequence number. All messages sent after a local snapshot recording are tagged by a tuple $< init_id, MKNO >$, where $init_id$ is the initiator's identifier and $MKNO$ is the sequence number of the algorithm's most recent invocation by $init_id$; markers with tags similar to the above tags are explicitly sent only on all outgoing channels on which no messages might be sent. Thus, the tuple $< init_id, MKNO >$ is either piggybacked on computation messages (implicit markers) or sent as explicit markers on each outgoing channel after recording the local state.

A process keeps track of the sequence number of the last snapshot and Input/Output

The algorithm requires each process i to maintain arrays $SENT_i[1, ...N]$ and $RECD_i[1, ..., N]$. $SENT_i[j]$ 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. However, the arrays do not contribute to the storage complexity of the algorithm because causal ordering protocols require these arrays. The algorithm works as follows: An initiator process multicasts 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, which includes the local state, $RECD_i$ and $SENT_i$, when it receives $token_i$ and sends the recorded snapshot to the initiator. The algorithm terminates when the initiator receives the snapshot recorded by each process. In the global snapshot assembled by the initiator, the state of each channel from the initiator to each process is empty and the state of any other channel from process i to process j is the set of messages whose sequence numbers are given by $\{RECD_j[i] + 1, \ldots, SENT_i[j]\}$.

The algorithm does not require each process to send markers on each channel, and the processes do not coordinate their local snapshot recordings with one another. In spite of this, for any i and j which record their local snapshots LS_i and LS_j, respectively, $inconsistent(LS_i, LS_j) = \emptyset$ due to the causal ordering property of the underlying system. Let a message 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 support ensures that $rec(token_j)$, at which instant j records LS_j, happens before $rec(m_{ij})$. Thus, m_{ij} which is not recorded as sent in LS_i, is not recorded as received in LS_j and $inconsistent(LS_i, LS_j) = \emptyset$.

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.

5 Snapshot Algorithms for NonFIFO Systems

A FIFO system ensures that all messages sent after a marker will be delivered after the marker. This ensures that $inconsistent(LS_i, LS_j) = \emptyset$ in the recorded snapshot if LS_i and LS_j are recorded as described in the Chandy-Lamport algorithm, and its variants outlined above. In a non-FIFO system, different techniques have to be used to ensure that $inconsistent(LS_i, LS_j) = \emptyset$ because the marker cannot play the role of delineating of messages into those that have to be recorded in the global state and those that are not to be recorded in the global state. In a non-FIFO system, some degree of message inhibition, or process blocking, is necessary to record a consistent global snapshot [9]. The non-FIFO algorithm proposed by Helary uses message inhibition [4]. The algorithms by Lai and Yang [5], Li et al. [6], and Mattern [3] do not use message inhibition; however, these algorithms superimpose a FIFO ordering for a marker with respect to computation messages sent after the marker, even though the underlying system is non-FIFO.

- it receives a marker for this snapshot from j, or
- it is notified of the next wave by a control message belonging to the next wave.

This algorithm has a message complexity of $O(e)$ to record a snapshot; additional messages are needed to implement the wave.

4.4 Snapshots in a Causal Delivery System

Acharya and Badrinath [1] presented an algorithm for recording snapshots in a system which supports causal message delivery. The algorithm does not need to send markers on every channel and is simpler than snapshot algorithms in a FIFO system because the strong causal delivery property ensures that $inconsistent(LS_i, LS_j) = 0$ for every LS_i and LS_j recorded as part of the global snapshot, as explained subsequently.

Begin ——————————————————————————SIDEBAR 1

Causal ordering was first proposed by Birman and Joseph [1] and was implemented in the ISIS project. The causal order of message delivery is defined as follows: For any two messages m_{ij} and m_{kj}, if $send(m_{ij}) \longrightarrow send(m_{kj})$, where $\longrightarrow$ is Lamport's *happens before* [1] relation [3], then $rec(m_{ij}) \longrightarrow rec(m_{kj})$. It follows from the defintion of causal ordering that FIFO message delivery is ensured. Techniques for causal ordering of messages are useful in developing distributed algorithms and may simplify the algorithms themselves. For example, in replicated database systems, it is important that every process responsible for updating a replica receives the updates in the same order to maintain database consistency. Without causal ordering, each update must be checked to ensure that database consistency is not being violated. Causal ordering eliminates the need for such checks.

There are several known protocols for implementing causal ordering [1, 4, 2].

References

[1] K. Birman, T. Joseph, *Reliable Communication in Presence of Failures,* ACM Transactions on Computer Systems, 47-76, 3, 1987.

[2] K. Birman, A. Schiper, P. Stephenson, *Lightweight Causal and Atomic Group Multicast,* ACM Transactions on Computer Systems, 272-314, 9(3), 1991.

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

[4] A. Schiper, J. Eggli, A. Sandoz, *A New Algorithm to Implement Causal Ordering,* Proceedings of the 3rd International Workshop on Distributed Algorithms, LNCS 392, Springer Verlag, 219-232, 1989.

[1] Event e_1 *happens before* event e_2 if (a) e_1 occurs before e_2 on the same process, or (b) e_1 is an event at which a message is sent and e_2 is the event at which the message sent at e_1 is received, or (c) $\exists e' |$ e_1 happens before e' and e' happens before e_2.

when this condition holds for the initiator (a non-leaf process). The selective manner in which *regular* markers are sent has the effect that a process does not know whether to expect a *regular* marker on an incoming channel. A process can be sure that no such marker will be recieved and that the snapshot is complete only when it executes the "Marker Sending Rule" for the next version of the algorithm.

Venkatesan [10] 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 computational message has been sent since the previous snapshot. The above incremental snapshot algorithm achieves this lower bound in message complexity.

4.3 Helary's Wave Synchronization Method

Helary proposed a wave synchronization-based method for recording global snapshots [4]. 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.

In Helary's algorithm, the "Marker Sending Rule" is executed when a control message belonging to the wave flow visits the process. The process then forwards a control message(s) to other process(es), depending on the wave traversal structure, to continue the wave's progression. The "Marker Receiving Rule" is modified so that if the process has not recorded its state at the time a marker is received on some channel, the "Marker Receiving Rule" is not executed and no messages received on this channel are processed until the control message belonging to the wave flow visits the process. Thus, each process follows the "Marker Receiving Rule" only after it is visited by a control message belonging to the wave.

The primary function of wave synchronization is to evaluate functions over the recorded global snapshot. An example of this function is the number of messages in transit to each process in a global snapshot, and whether the global snapshot is strongly consistent. For this function, each process maintains two vectors, $SENT$ and $RECD$. The i^{th} elements of these vectors indicate the number of messages sent to/received from process i, respectively, since the previous visit of a wave control message. The wave control messages carry a global abstract counter vector whose i^{th} entry indicates the number of messages in transit to process i. These entries in the vector are updated using the $SENT$ and $RECD$ vectors at each node visited. When the control wave terminates, the number of messages in transit to each process as recorded in the snapshot is known.

This algorithm is extended to non-FIFO systems by imposing two additional rules to ensure that $inconsistent(LS_i, LS_j) = \emptyset$. First, when a marker is received, an acknowledgement is returned. Second, once a process i executes the "Marker Sending Rule", it does not send any messages on an outgoing channel C to j until it is sure that j has recorded its local state. Process i can conclude this if:

- it receives an acknowledgement for the marker sent to j, or

$id - border - set$) to its parent in the spanning tree. When an intermediate process in a spanning tree receives the recorded states from all its child processes and has recorded the states of all incoming channels, it forwards its locally recorded state and the locally recorded states of all its descendant processes to its parent.

When the initiator receives the locally recorded states of all its descendants from its child processes, it assembles the snapshot for all the processes in its region and the channels incident on these processes. The initiator knows the identifiers of initiators in adjacent regions using $id - border - set$ information it receives from processes in its region. The initiator exchanges the snapshot of its region with the initiators in adjacent regions in rounds. In each round, an initiator sends to initiators in adjacent regions, any new information obtained during the previous round of message exchange. A round is complete when an initiator receives information, or the *blank* message (signifying no new information will be forthcoming) from all initiators of adjacent regions from which it has not already received a *blank* message.

The message complexity of recording is $O(e)$, which is that required by one initiator, even if there are r concurrent initiators . The message complexity of assembling and disseminating the snapshot is $O(rn^2)$.

4.2 Message-Optimal Incremental Snapshots

Many of the applications requiring global snapshots need successive snapshots of the system. For example, recovery algorithms with synchronous checkpointing need to advance their checkpoints periodically. This can be achieved by using a protocol that uses a lesser complexity than that of the repeated invocations of the Chandy-Lamport algorithm. The latest snapshot of the system can be obtained by running a protocol to record an incremental snapshot since the most recent snapshot was assembled, and combining its result with the most recent snapshot. The incremental snapshot algorithm proposed by Venkatesan [10] modifies the global snapshot algorithm of Chandy-Lamport to save on message complexity when messages of the underlying computation are sent on only a few of the network edges between two successive snapshots.

The incremental snapshot algorithm assumes bidirectional FIFO channels, the presence of a single algorithm initiator, a fixed spanning tree in the network, and four types of markers: *init_snap* markers, *snap_completed* markers, *regular* markers, and *ack* markers. The algorithm works as follows: Snapshots are assigned version numbers and all algorithm messages carry this version number. The initiator notifies all the processes the version number of the new snapshot by sending *init_snap* markers along the spanning tree edges. A process follows the "Marker Sending Rule" when it receives this notification or when it receives a *regular* marker with a new version number. The "Marker Sending Rule" is modified so that the process sends *regular* markers only along those edges on which it has sent computation messages since the previous snapshot, and the process waits for *ack* markers in response to these *regular* markers. When a leaf process in the spanning tree receives all the *ack* markers it expects, it sends a *snap_completed* marker to its parent process. When a non-leaf process in the spanning tree receives all the *ack* markers it expects, as well as a *snap_completed* marker from each of its child processes, it sends a *snap_completed* marker to its parent process. The algorithm terminates

A physical interpretation of the collected global state is as follows: Consider the two instants of recording the local states in the banking example. These instants are marked by 'X's in Figure 4. If the cut formed by these instants is viewed as being an elastic band, then the recorded global state occurs in the execution that results if the elastic band is stretched so that it is vertical.

Several algorithms for recording global snapshots have been proposed after the Chandy-Lamport algorithm.

4 Other Snapshot Algorithms for FIFO Systems

4.1 Spezialetti-Kearns Method

Spezialetti and Kearns [11] provided two optimizations to the Chandy-Lamport algorithm. The first optimization combines snapshots concurrently initiated by multiple processes into a single snapshot. This optimization is linked with the second optimization which deals with the assembly and distribution of the global snapshot. This algorithm assumes bidirectional channels in the system.

In the Spezialetti-Kearns algorithm, markers carry an identifier for the initiator of the algorithm. Each process has a variable *master* to keep track of the initiator of the algorithm. When a process executes the "Marker Sending Rule" due to the receipt of its first marker, it records the initiator's identifier carried in the received marker in the *master* field. A process that initiates the algorithm records its own identifier in the *master* field.

A key notion used by the optimizations to the Chandy-Lamport algorithm is that of a *region* in the system. All the processes whose *master* field contains the identifier of the same initiator lie in a region which is identified by the initiator's identifier. When there are multiple concurrent initiators, the system gets partitioned into multiple regions.

When the initiator's identifier in a marker received along channel C is different from the value of *master*, a concurrent initiation of the algorithm is detected, and the sender of the marker lies in a different region. The identifier of the concurrent initiator is recorded in a local variable $id-border-set$. The state of C is recorded just as in the Chandy-Lamport algorithm. This enables the snapshot recorded in one region to be merged with the snapshot recorded in the adjacent region. Thus, even though markers arriving at a node contain identifiers of different initiators, they are considered part of the same instance of the algorithm.

The Spezialetti-Kearns algorithm records states of processes and channels within a region just as in the Chandy-Lamport algorithm. The snapshot of the entire system is recorded because the states of channels that cross a border between regions are also recorded. The mechanism for assembling the snapshot is as follows: In the snapshot recording phase, a forest of spanning trees is implicitly created in the system. The initiator of the algorithm is the root of a spanning tree. All processes in its region belong to its spanning tree. The parent of a process i is process j if process i executed the "Marker Sending Rule" because it received its first marker from process j. When a leaf process in the spanning tree has recorded the states of all incoming channels, the process sends the locally recorded state (local snapshot,

markers. Different initiations by a process are identified by a sequence number.

The recorded local snapshots can be put together to create the global snapshot in several ways. One policy is for each process to send its local snapshot to the initiator of the algorithm. Another policy is for each process to send the information it records along all outgoing channels, and for each process receiving such information for the first time to 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. However, there could be upto n instances of the algorithm in progress at any time.

3.1 Property of the Recorded Global State

The recorded global state may not correspond to any of the global states that occurred during the computation. We will consider a possible execution of the global snapshot algorithm for the money transfer example of Figure 1 using a timing diagram in Figure 2. Let Site 1 initiate the algorithm at the end of step 1. Site 1 records its local state (Account A = \$500) and sends a marker to Site 2. The marker is received by site 2 at the end of step 4. When site 2 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 A 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 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, the recorded global state is a meaningful global state because the system could have passed through that (consistent) global state in an equivalent execution [2]. Suppose the algorithm is initiated in global state S_i and it terminates in gobal 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 [2] 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'. The proof runs as follows: An event e is defined as a prerecording/postrecording event if e occurs on a process p and p records its state after/before e in seq. A postrecording event may occur after a prerecording event only if the two events occur on different processes. It is shown that a postrecording event can be swapped with an immediately following prerecording event in a sequence without affecting the local states of either of the two processes on which the two events occur. By iteratively applying this operation to seq, the above-described permutation seq' is obtained. It is then shown that S^*, the global state recorded by the algorithm for the processes and channels, is the state in seq' after all the prerecording events and before all the postrecording events. The recorded global state is useful for detecting a stable property, i.e., a property that persists, of the system, such as termination or deadlock. If a stable property holds before the snapshot algorithm begins, it continues to hold and gets recorded in the global snapshot.

Marker Sending Rule for process i

1. Process i records its state.
2. 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.

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.

3 The Chandy-Lamport Algorithm

Chandy and Lamport were the first to formulate the notion of a consistent global state and present an algorithm to record it [2]. The algorithm assumes FIFO communication channels and uses special messages called *markers* that have meaning only to the snapshot algorithm. The algorithm is given in Figure 3.

An initiator process triggers 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 recording the local state and before receiving the marker on that channel.

The role of the 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 $l \geq m$. A marker delineates messages into those that have to be recorded in the global state and those that are not to be recorded in the global state. The algorithm can be initiated by any process by executing the "Marker Sending Rule". If multiple processes initiate the algorithm concurrently, each initiation needs to be distinguished by using unique

- $rec(m_{ij}) \in LS_j$ iff $time(rec(m_{ij})) < time(LS_j)$

For a pair of processes i and j with a channel from i to j, two sets of messages can be defined based on their local states [4].

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

Inconsistent: $inconsistent(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \notin LS_i \bigwedge rec(m_{ij}) \in LS_j \}$

2.1 Global State

A global state is a collection of the local states of the processes and channels. The state of a process includes the messages sent/received on each incident channel. A channel is a distributed entity and its state depends on the local states of the processes on which it is incident. Hence, a global state can be defined as $GS = \{LS_1, LS_2, \ldots, LS_n\}$ where n is the number of sites in the system. The state of a channel from process i to process j in GS is defined as $transit(LS_i, LS_j)$.

A global state GS is a *consistent global state* iff

$$\forall i, \forall j \ : 1 \leq i, j \ \leq n \ :: inconsistent(LS_i, LS_j) = \phi.$$

In a consistent global state, every message that is recorded as received is also recorded as sent and captures the notion of causality that a message cannot be received if it is not sent. Consistent global states are meaningful global states and inconsistent global states are not meaningful.

A global state is *transitless* iff

$$\forall i, \forall j \ : 1 \leq i, j \ \leq n \ :: transit(LS_i, LS_j) = \phi.$$

Thus, all channels are empty in a transitless global state. Note that the transitlessness of a global state implies its consistency. Transitlessness not only implies that all the received messages have been sent but also that all the messages sent have been received.

A global state is *strongly consistent* iff it is transitless. These states have special significance in recovery of distributed systems. Helary [4] has provided a way to record a strongly consistent state.

2.2 Cuts

A distributed computation can be conveniently represented using a timing diagram where horozontal lines represent the processes' time lines [3]. 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. A consistent global

1. Account A = \$400, Account B = \$150, C1 = \$100, C2 = \$50.

4. Site A receives the message for a \$50 credit to Account A and updates Account A. Account A = \$450, Account B = \$150, C1 = \$100, C2 = \$0.

5. Site B receives the message for a \$100 credit to Account B and updates Account B. Account A = \$450, Account B = \$250, C1 = \$0, C2 = \$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 C1 and C2 are recorded at the end of step 3 to show \$150, \$ 100, and \$50, respectively. Then the composite of all the states 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 C1 was initiated, whereas Channel C1's state was recorded after the \$100 transfer was initiated. Therefore, a recorded global state may be inconsistent if $l < l'$, where l is the number of messages sent by A along the channel before A's state is recorded, and l' is the number of messages sent by A along the channel before the channel's state is recorded.

Suppose the local state of C1 is recorded at the end of step 1 to show \$0 and the local state of Accounts A and B, and C2 is recorded at the end of step 3 to show \$400, \$150, and \$50, respectively. Then the composite of all the states shows \$600 in the system. A deficit of \$100 appears in the system. The reason for the inconsistency is that Account A's state was recorded after the \$100 transfer to Account B using Channel C1 was initiated, whereas Channel C1's state was recorded before the \$100 transfer was initiated. Thus, the recorded global state may be inconsistent if $l > l'$. Therefore, for a consistent global state, it is necessary that $l = l'$.

It can be similarly shown that it is necessary to have $m = m'$, where m is the number of messages received along the channel by B before Account B's state was recorded and m' is the number of messages received along the channel by B before the channel's state was recorded.

In every system, the number of messages sent along a channel is never less than the number of messages received along that channel. Therefore, $l' \geq m'$, hence for consistency,

$$l \geq m$$

Any recorded global state must satisfy this constraint [2]. Next we formalize these concepts.

2 Definitions

The local state of a process at any time is defined by the state of the local memory associated with the local context of the distributed application. Let the local state of process i at any time be denoted by LS_i. Let m_{ij} denote a message sent by process i to process j. Let $send(m_{ij})$ denote the send event of a message m_{ij} and let $rec(m_{ij})$ denote the receive event of a message m_{ij}. Let $time_x$ denote the time at which event x occurred. For a message m_{ij}, it can be said that

- $send(m_{ij}) \in LS_i$ iff $time(send(m_{ij})) < time(LS_i)$

1 Introduction

A distributed computing system consists of spatially separated processes that do not share a common memory and that communicate with each other by message passing over communication channels. The system can be described as a directed graph in which vertices represent the processes and edges represent 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 channnel. The global state of a distributed system is a collection of the local states of all its components.

The global state of a system is important for monitoring distributed events, setting distributed breakpoints [7], protocol verification [3], detection of stable properties such as deadlocks and termination, checkpointing, debugging distributed software, and discarding obsolete information in databases [8]. Therefore, it is important that there be efficient ways of observing the global state of a distributed system. Unfortunately, there is no shared memory and no global clock in a distributed system; the distributed nature of the local clocks and local memory makes it difficult to observe 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 is possible if the local clocks can be perfectly synchronized or if there is a global system clock. However, the clock value at each site is different because it is technologically infeasible to perfectly synchronize clocks. If the various processes read time from a single common (global) clock, various indeterminate transmission delays during the read operation will cause the processes to identify various physical instants at 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 1 to Site 2 and from Site 2 to Site 1 be denoted by C1 and C2, respectively. Consider the following sequence of actions, which are also illustrated in Figure 1:

1. Initially, Account A = \$500, Account B = \$200, C1 = \$0, C2 = \$0.

2. Site A 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 C1 to Site 2. Account A = \$400, Account B = \$200, C1 = \$100, C2 = \$0.

3. Site B 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 C2 to Site

Global Snapshots of a Distributed System

Ajay D. Kshemkalyani
IBM Corporation
P. O. Box 12195
Research Triangle Park
NC 27709

Michel Raynal
IRISA
campus de beaulieu
35042 Rennes-cedex
France

Mukesh Singhal
Department of Computer
and Information Science
The Ohio State University
Columbus, OH 43210

Abstract

The global state of a distributed system is an important issue in the design of distributed systems and it is important that there be efficient ways of observing the global state of a distributed system. Unfortunately, the lack of both a globally shared memory and a global clock in a distributed system makes it difficult to observe the global state of the system. In this paper, we present several algorithms to collect a snapshot of a distributed system and compare their features.

Keywords: global state, snapshot, distributed system, stable property.