Lazy Snapshots

Nigamanth Sridhar and Paolo A.G. Sivilotti
Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277
{nsridhar,paolo}@cis.ohio-state.edu

Abstract

Determining the global state of distributed systems is an important problem in the absence of global memory and global clocks. Several algorithms have been proposed for collecting global states. This paper presents an optimization to the algorithm proposed by Chandy and Lamport that lets processes in a distributed system take a lazy approach to recording their state. The lazy optimization relaxes the constraint on when a process must record its local state, and send out markers. Our algorithm yields the flexibility of postponing a local snapshot and hence reducing the amount of space required to save channel state.

Keywords: parallel/distributed algorithms, snapshots, termination detection.

1 Introduction

The global state of a distributed system consists of the state of each process and the messages in each channel. Recording the global state of a distributed system is an important problem and it finds applications in several aspects of distributed system design. Some such applications include the detection of stable properties such as deadlock [9] and termination [12]; checkpoint recovery from failures [8]; debugging distributed software, by resetting the system state to a consistent global state and restarting execution from that state [5, 6]; and transformation of an algorithm for solving a static network problem into one that solves the dynamic version of the same problem [3]. A global snapshot algorithm determines the global state of a system.

Several algorithms have been proposed to solve the problem of determining the global state of a distributed system [4, 10, 11, 13]. Common among all the algorithms is some way of distinguishing events that occurred *before* the snapshot from those that occurred *after* the snapshot. The most popular algorithm for the system model that we consider, described in Section 2, is the marker algorithm proposed by Chandy and Lamport [4].

The Chandy-Lamport algorithm uses marker messages that are sent by some process in the system identified as the initiator to distinguish the *before* messages from the *after* messages. In this paper, we present an optimization to the Chandy-Lamport algorithm that allows processes to take a *lazy* approach to recording their local state.

Section 2 describes the system model that we consider. The Chandy-Lamport algorithm is presented in Section 3. Sections 4 and 5 present our optimization and a proof that the algorithm is correct. We examine some related work in Section 6 and summarize our contributions and conclude in Section 7.

2 The System Model

A distributed system consists of a finite set of processes and a finite set of channels. It is described by a labeled, directed graph in which the vertices represent the processes and the edges represent the channels. There is no globally shared memory or clock. Processes communicate only through messages passed on channels that connect them. Let C_{pq} denote the channel from process p to process q.

Processes can perform three kinds of actions - internal events, message *send* events and message *receive* events. A message m_{pq} from process p to process q is sent by the action $send(m_{pq})$ executed on process p, and is received by the action $recv(m_{pq})$ executed on process q.

Channels are assumed to have infinite buffers, to be error-free, and to deliver messages in the order they are sent (FIFO). The message delay in a channel is finite. A message is said to be in *transit* when it has been sent by the source process but has not been received by the destination process. *i.e.*, $send(m_{pq})$ has been executed on process p, but $recv(m_{pq})$ has not been executed on process q. The state of a channel at a particular time is the set of all messages that are in transit in that channel.

The *global state* of the system is defined as the union of local states of all processes and the states of all channels. In a *consistent* global state, every message that is recorded as received has also been recorded as sent. Consistent global states are meaningful global states and inconsistent global states are meaningless, *i.e.*, the system could never be in such a state. Figure 1

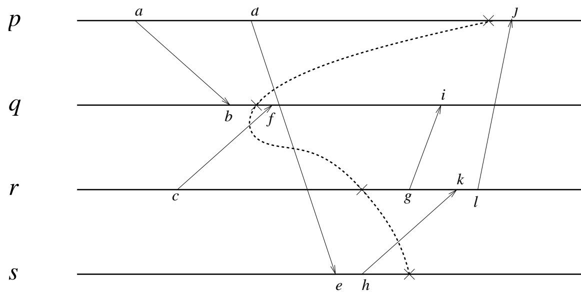

Figure 1. A consistent cut in a distributed system

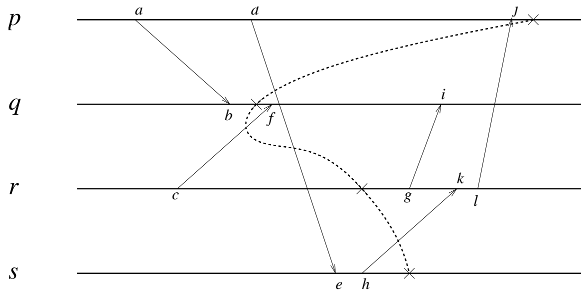

Figure 2. An inconsistent cut in a distributed system

shows a time-line diagram of a distributed process system with a consistent cut. Figure 2 shows a time-line diagram of a distributed system with an inconsistent global state.

3 The Chandy-Lamport Algorithm

This section presents the Chandy-Lamport algorithm for determining global states of distributed systems. The algorithm works as follows: Each process records its own state and the two processes on which a channel is incident cooperate in recording the channel state. The algorithm cannot ensure that the states of all processes and all channels are recorded at the same instant in time. However, the algorithm does ensure that the recorded process and channel states form a meaningful global system state.

The global state recording algorithm is superimposed with the underlying computation, *i.e.*, it runs concurrently with, but does not alter, the underlying computation. The algorithm sends messages and requires processes to carry out computations, but these do not interfere with the underlying computation.

The algorithm uses a marker to distinguish between events that occurred before a snapshot is taken and events that occurred after. The algorithm can be initiated by any process, identified as the initiator. The initiator spontaneously records its state and starts executing the algorithm. The outline of the algorithm in the form of rules is presented in Figure 3.

The global state gathered by this algorithm is

Marker-Sending Rule for a Process p. For each outgoing channel C:
p sends one marker along C after p records its state and before p sends further messages along the same channel.

Marker-Receiving Rule for a Process q.
On receiving a marker along a channel C:
if q has not recorded its state **then**
  begin q records its state;
    q records the state of C as the empty sequence
  end
else q records the state of C as the sequence of messages received along C after q's state was recorded and before q received the marker along C.

Figure 3. Chandy-Lamport algorithm for distributed snapshots

a consistent one. All the messages that have been recorded as received are also recorded as sent, *i.e.*, there are no *orphan* messages. Further, all messages that have been recorded as sent are either recorded as received at the destination process, or are recorded as being in transit in the channel that the message was sent along. See [4] for a full proof of correctness.

4 An Optimization: Lazy Snapshots

4.1 Overview

The algorithm presented in Section 3 requires that every process q, immediately on receiving a marker from another process p, takes a local snapshot. This ensures that no messages sent by p to q after p recorded its snapshot are included in q's snapshot. Indeed, *no messages* received after the first marker are included in q's local state. This condition is stronger than necessary: To maintain the consistency of the global state, we only need to guarantee that no *orphan* messages are included.

The optimization that we propose, therefore, allows a process to postpone recording of its local snapshot. The postponement, however, needs to be controlled so as to still maintain consistency of the global snapshot that is gathered.

The new algorithm works as follows. On receiving a marker from process p, process q "remembers" the reception of a marker from p. It sends markers on all outgoing channels as usual. However, q does not need to record its local snapshot as yet. It postpones the recording of the local snapshot to a later time. q is forced to take a local snapshot only if q receives a message from a process p, from which it has already received a marker.

By delaying the recording of a local snapshot, the

number of in-transit messages is decreased. Thus, a process can reduce the amount of channel state that it needs to record with the snapshot. The ability to postpone recording local state also has the advantage of giving a process flexibility in scheduling this potentially expensive task.

There is one technical problem with the postponement as described, however. Consider the case of a process r that does not communicate with the rest of the system. This process could just perform some local computation, never sending or receiving messages to the other processes. In such a case, all other processes in the system could take their local snapshots, but the global snapshot cannot be calculated until r records its local state.

In order to force the global state collector to terminate, a third event can be added: A marker has been received on every incoming channel. The local snapshot triggered by this event will record the state of every incoming channel as empty.

The global state that this algorithm collects is indeed consistent. The algorithm can be seen as a generalization of the Chandy-Lamport algorithm. It reduces the space complexity of the recorded channel state and permits flexibility in scheduling the potentially expensive task of recording local state. The complete algorithm is presented in Section 4.3 and Section 5 proves that the algorithm is correct.

4.2 The Snapshot Layer

The snapshot collector is composed with the system by layering. We call the original program the *system layer* and the snapshot collector executes in the *snapshot layer*. As a result of the composition, each process in the system is associated with a *snapshot proxy*. This proxy cooperates with the proxies of the rest of the processes in the system to compute the global state of the distributed system.

The snapshot proxy running in the snapshot layer does not affect the system layer in any way. However, each proxy does examine all messages that its associated process sends or receives. Based on markers that are passed around in the snapshot layer and the messages in the system layer that the proxy examines, it sends a message to its host process asking it to take a local snapshot. In this way, the processes in the system layer need no knowledge of the snapshot collector protocol.

The snapshot proxy has some state associated with it in order to be able to make decisions of when to instruct its host process to record its local state. The proxy is responsible for keeping track of which channels are *dirty* (a marker has been received along that channel), and when local state has been recorded.

4.3 The Algorithm

The algorithm for lazy snapshots is presented in Figure 4 in the form of rules. The snapshot collector will execute concurrently with the underlying program.

Marker-Sending Rule for a Process p. For each outgoing channel C:
p sends one marker along C after p records its state and before p sends further messages along the same channel.

Marker-Receiving Rule for a Process q.
On receiving a marker along a channel C:
mark channel C as *dirty*;
if q has not recorded its state **then**
q records the state of C as the empty sequence
else q records the state of C as the sequence of messages received along C after q's state was recorded and before q received the marker along C.

State Recording Rule for a Process p. Process p records its state after receiving a marker and before p receives any message along a *dirty* channel.

Figure 4. Lazy Snapshot algorithm

When process q receives a marker from any neighbor p, it "remembers" such a receipt by marking the incoming channel C as *dirty*. Whenever a process p receives a message from a process q, p checks if the corresponding channel is *dirty*. If it is, then p records its local snapshot, and then proceeds to process the message from q. This is because the current message from q has not been recorded as sent in the local snapshot of q. (The FIFO behavior of the channel guarantees that the message was sent after the marker was sent.) If p records this message as received, then this would lead to an inconsistent global state.

After p records its local state, if p receives a marker from one of its neighbors, it needs to record the state of the channel C on which p received the marker. The state of the channel will be the sequence of messages that p received along C in the time between the recording of p's local state and the reception of the marker on C.

If p has received markers from all of its neighbors, and still has not taken a local snapshot, p records its local state at this point. A snapshot taken at this time will include both local state and the state of all the incoming channels, all empty. This is because p has received markers from all its neighbors, and hence any future message it receives from *any* of its neighbors should not be in the snapshot for it to be consistent.

5 Proof of Correctness

5.1 Notation

Before we present the proof of correctness of the lazy snapshot algorithm, we present here some notation, and a characterization of the rules in the algorithm.

We use the following terms to refer to the (absolute) time at which the principal actions occur:

RLS_p **:** Process p records its local state.

$RM_{C_{pq}}$ **:** Process q receives a marker on channel C_{pq} .

$SM_{C_{pq}}$ **:** Process p sends a marker on channel C_{pq} .

$RD_{C_{pq}}$ **:** Process q receives a message on a dirty channel C_{pq} .

$US_{C_{pq}}$ **:** Process p sends a message on channel C_{pq} *after* recording local state (an "unrecorded send").

Using the terms presented above, the original algorithm proposed by Chandy and Lamport can be characterized by the following inequalities:

E1. $(\forall p :: RLS_p \leq (\textbf{Min}\, q :: RM_{C_{qp}}))$

E2. $(\forall p, q :: RLS_p < SM_{C_{pq}} < US_{C_{pq}})$

The inequality **E1** says that a process p must record its local state either before or—at the latest–at the time it receives its first marker. This is the same as the Marker Sending Rule in the original algorithm, which states that each process must record its local state upon receiving the first marker along any of its incoming channels. **E1** also includes the case of the initiator process, which records its local state before any markers are received.

The second inequality **E2** requires that each process p must send a marker along each of its outgoing channels *after* recording its local state and *before* sending any messages along that particular channel. That is, any unrecorded send along any channel C_{pq} must be preceded by a marker along that same channel. Further, this marker has to be sent only after the local state of the process has been recorded. This is true of all processes, including the initiator process, which sends out markers to its neighbors after recording its own state.

Similarly, the lazy snapshot algorithm can also be characterized by a pair of inequalities:

L1. $(\forall p :: RLS_p < (\textbf{Min}\, q :: RD_{C_{pq}}))$

L2. $(\forall p, q :: SM_{C_{pq}} < US_{C_{pq}})$

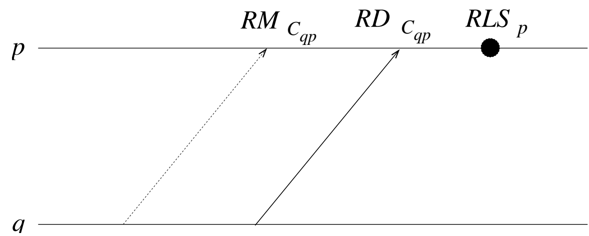

Figure 5. Receive on a dirty channel preceding RLS – inconsistent cut

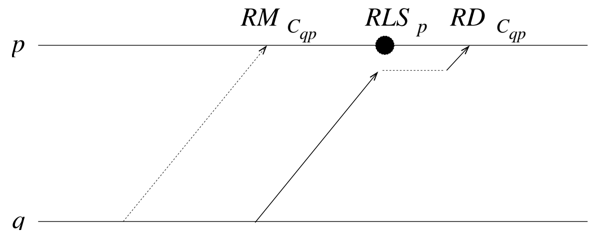

Figure 6. Receive on a dirty channel after RLS – consistent cut

L1 requires that each process p in the network record its local state before the first message along any dirty channel is received. Figure 5 shows the receipt of a message on a *dirty* channel before the local state has been recorded. This would result in an inconsistent global state. Figure 6 shows the receipt of such a message being preceded by a local snapshot, which is what **L1** enforces. This is consistent with the State Recording Rule in the lazy snapshot algorithm (Figure 4). This is a relaxation of **E1**, in that processes are not forced to record their local state upon receipt of markers. Instead, the local state recording can be postponed until the first message along a dirty channel is received.

L2 is a relaxation of **E2**. We still require that before a process sends an unrecorded message along any channel, a marker must have been sent along that same channel. This is how we can ensure that no inconsistencies occur in the global snapshot. Figure 7 shows an unrecorded send at a process p before a marker has been sent along the same channel. This would result in an orphan message figuring as part of the global state. This is rectified in Figure 8, where the unrecorded message is sent after a marker has been sent along the same channel. This marker would force process q to take its local snapshot without including the receipt of the current message. However, the sending of markers is not dependent on when the local state is recorded. A process is free to record its local state at any time, regardless of when its sends markers along its outgoing channels. The recording of local state is governed only by **L1**.

For any process p, $(\forall q :: RM_{C_{qp}} < RD_{C_{qp}})$.

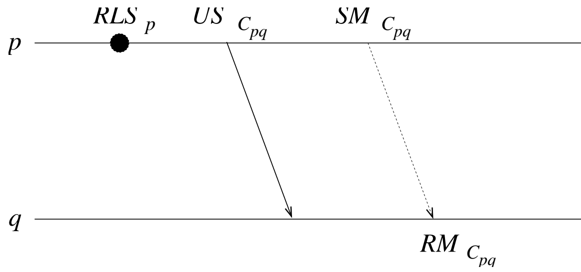

Figure 7. Unrecorded send preceding marker in channel C_{pq} – inconsistent cut

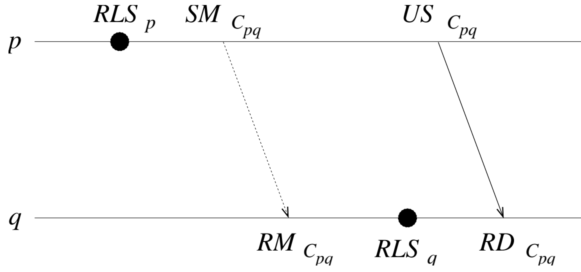

Figure 8. Unrecorded send in channel C_{pq} after marker has been sent – consistent cut

Therefore, **E1** $\Rightarrow$ **L1**. Further, **E2** $\Rightarrow$ **L2**. These two implications show how the lazy snapshot algorithm is a generalization of the original formulation of the global state detection algorithm.

5.2 Proof

The safety and progress properties that a correct global state detection algorithm should have are as follows:

Safety. The global state collected by the algorithm is legal (consistent cut)

Progress. The global state detection algorithm terminates

In order to prove the safety part, we have to show that no orphan messages are part of the global snapshot. This requires us to show that the following proof obligations are indeed properties of the lazy snapshot algorithm:

S1. Every message recorded as received has been recorded as sent.

S2. Every message recorded as in transit has been recorded as sent.

Proof. In order for **S1** to be violated, a process p has to have sent a message m_{pq} after its local snapshot (the message is not recorded as sent in the global state), and q does record the receipt of m_{pq} in its local state. In order for m_{pq} to be an unrecorded send, p must have already sent a marker along C_{pq}. This marker, upon reaching q, makes the channel C_{pq} *dirty* from q's perspective. This means that before q can process the receipt of any message along this channel, it has to record its local state (**L1**). Since the system model assumes FIFO channels, and m_{pq} arrives at q along a dirty channel, it is not recorded as received in q's local state, and consequently in the global state. Thus, **S1** is a property of the lazy snapshot algorithm.

A message m_{pq} is marked as being in transit by a process q if the message arrives after q has taken its local snapshot. In fact, this message is included as "in transit" only after a marker is received along the same channel is received. Since channels are FIFO, the marker along C_{pq} must have been sent by p after the message m_{pq} was sent. This can only be the case if the send of m_{pq} has indeed been recorded in p's local snapshot. If, on the other hand, this had been an unrecorded send, the send must have been preceded by a marker along the channel C_{pq} (**L2**). Thus, **S2** is also a property of the lazy snapshot algorithm. □

The progress part of the specification states that the snapshot collector eventually computes the global state of the system. The proof that the lazy snapshot algorithm indeed terminates follows.

Proof. The algorithm begins with an initiator process sending off a marker to each of its neighbors to initiate the collection of a new snapshot. According to the Marker-Receiving Rule, each process, on receiving a marker along an incoming channel remembers the receipt of the marker, and also forwards the marker on all its outgoing channels. The marker, therefore, gets propagated to all the processes in the system. Thus, every process eventually receives a marker along every one of its incoming channels. The Marker-Receiving Rule also checks to see if this particular marker is the last marker, and if it is, the process takes its local snapshot. Since all processes will receive markers along all incoming channels, all processes will take their local snapshots. That is, the snapshot collector will eventually compute the global state of the distributed system. □

6 Related Work

Several solutions to the global state detection problem have been proposed. Spezialetti and Kearns [14] proposed an optimization of the Chandy-Lamport algorithm to combine concurrently initiated snapshots. This way, if multiple processes initiate snapshot windows concurrently, the processes will only need to take

one local snapshot and distribute the same local snapshot to the different initiators.

Venkatesan [15] proposed an incremental approach to collecting global snapshots. Using this solution, each process maintains the most recent snapshot taken. A new local snapshot would then just involve combining the local state changes since the last snapshot with the most recent snapshot. This algorithm, however, assumes the presence of only a single initiator process.

Another extension to the Chandy-Lamport algorithm was proposed by Helary [7]. In this algorithm, snapshot windows are marked by using message waves. Every process in the system is visited by a wave control message, and this triggers the recording of local state at the process. As soon as a wave terminates, the next wave is initiated.

Several other solutions to the problem have been proposed under different system models such as non-FIFO systems [10, 11, 13] and causal ordering systems [1, 2].

Our algorithm differs from this body of work in that it generalizes the Chandy-Lamport scheme, yielding the flexibility of postponing a local snapshot and hence reducing the amount of space required to save channel state.

7 Conclusion

In this paper, we have presented an optimization to the marker-based global state detection algorithm proposed by Chandy and Lamport. The new algorithm allows processes in a distributed system to postpone their recording of local state up to a point where the global state is still consistent. In systems where the degree of the communication graph is high, and the time complexity of recording and updating local state is large, this optimization promises much better results by reducing the number of updates to the minimum possible.

8 Acknowledgments

This research was supported in part by NSF grant CCR-0081596, by an Ameritech Faculty Fellowship, and through a gift from Lucent Technologies.

References

[1] A. Acharya and B. R. Badrinath. Recording distributed snapshots based on causal order of message delivery. *Information Processing Letters*, 44(6):317–321, 1992.

[2] A. Alagar and S. Venkatesan. An optimal algorithm for distributed snapshots with causal message ordering. *Information Processing Letters*, 50:311–316, 1994.

[3] G. Bracha and S. Toueg. Distributed deadlock detection. *Dist. Comp.*, 2(3):127–138, 1987.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[5] R. Cooper and K. Marzullo. Consistent detection of global predicates. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):167–174, 1991.

[6] E. Fromentin, N. Plouzeau, and M. Raynal. An introduction to the analysis and debug of distributed computations. *Proc. of 1st Intl. Conf. on Algorithms and Architectures for Parallel Processing*, pages 545–553., May 1995.

[7] J.-M. Helary. Observing global states of asynchronous distributed applications. In *Proc. of the 3 rd Intl. Workshop on Dist. Algorithms*, number 392 in LNCS, pages 124–135. Springer, 1989.

[8] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. on Soft. Engg.*, 13(1):23–31, 1987.

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

[10] T. Lai and T. Yang. On distributed snapshots. *Inf. Proc. Letters*, 3(25):153–158, May 1987.

[11] H. F. Li, T. Radhakrishnan, and K. Venkatesh. Global state detection in non-FIFO networks. In *International Conference on Dist. Comp. Systems*, pages 364–370, 1987.

[12] F. Mattern. Algorithms for distributed termination detection. *Dist. Comp.*, 2(3):161–175, 1987.

[13] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18:423–34, 1993.

[14] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proc. 6th Int. Conf. on Dist. Computing Systems (ICSCS-6)*, pages 382–388, 1986.

[15] S. Venkatesan. Message-optimal incremental snapshots. *Journal of Computer and Software Engineering*, 1(3):211–231, 1993.