Chapter 9

Global Snapshot

9.1 Introduction

One of the difficulties in a distributed system is that no process has access to the global state of the system, that is, it is impossible for a process to know the current global state of the system (unless the computation is frozen). For many applications, it is sufficient to capture a global state that happened in the *past* instead of the *current* global state. For example, in case of a failure the system can restart from such a global state. As another example, suppose that we were interested in monitoring the system for the property that the token in the system has been lost. This property is *stable*, that is, once it is true it stays true forever; therefore, we can check this property on an old global state. If the token is found to be missing in the old global state, then we can conclude that the token is also missing in the current global state. An algorithm that captures a global state is called a *global snapshot algorithm*.

A global snapshot algorithm is a useful tool in building distributed systems. Computing a global snapshot is beautifully exemplified by Chandy and Lamport as the problem of taking a picture of a big scene such as a sky filled with birds. The scene is so big that it cannot be captured by a single photograph, and therefore multiple photographs must be taken and composed together to form the global picture. The multiple photographs cannot be taken at the same time instant because there is no shared physical clock in a distributed system. Furthermore, the act of taking a picture cannot change the behavior of the underlying process. Thus birds may fly from one part of the sky to the other while the local pictures are being taken. Despite these problems, we require that the composite picture be meaningful. For example, it should give us an accurate count of the number of birds. We next define what is meant by "meaningful" global state.

Consider the following definition of a global state: A *global state* is a set of local states that occur simultaneously. This definition is based on physical time. We use the phrase "time-based model" to refer to such a definition. A different definition of a global state based on the "happened-before model" is possible. In the happened-before model, a global state is a set of local states that are all concurrent with each other. By *concurrent*, we mean that no two states have a happened-before relationship with each other. A global state in the time-based model is also a global state in the happened-before model; if two states occur simultaneously, then they cannot have any happened-before relationship. However, the converse is not true; two concurrent states may or may not occur simultaneously in a given execution.

We choose to use the definition for the global state from the happened-before model for two reasons.

1. It is impossible to determine whether a given global state occurs in the time-based model without access to perfectly synchronized local clocks. For example, the statement "there exists a global state

in which more than two processes have access to the critical section" cannot be verified in the time-based model. In the happened-before model, however, it is possible to determine whether a given global state occurs.

2. Program properties that are of interest are often more simply stated in the happened-before model than in the time-based model, which makes them easier to understand and manipulate. This simplicity and elegance is gained because the happened-before model inherently accounts for different execution schedules. For example, an execution that does not violate mutual exclusion in the time-based model may do so with a different execution schedule. This problem is avoided in the happened-before model.

It is instructive to observe that a consistent global state is not simply a product of local states. To appreciate this, consider a distributed database for a banking application. Assume for simplicity that there are only two sites that keep the accounts for a customer. Also assume that the customer has \$500 at the first site and \$300 at the second site. In the absence of any communication between these sites, the total money of the customer can be easily computed to be \$800. However, if there is a transfer of \$200 from site A to site B, and a simple procedure is used to add up the accounts, we may falsely report that the customer has a total of \$1000 in his or her accounts (to the chagrin of the bank). This happens when the value at the first site is used before the transfer and the value at the second site after the transfer. It is easily seen that these two states are not concurrent. Note that \$1000 cannot be justified even by the messages in transit (or, that "the check is in the mail").

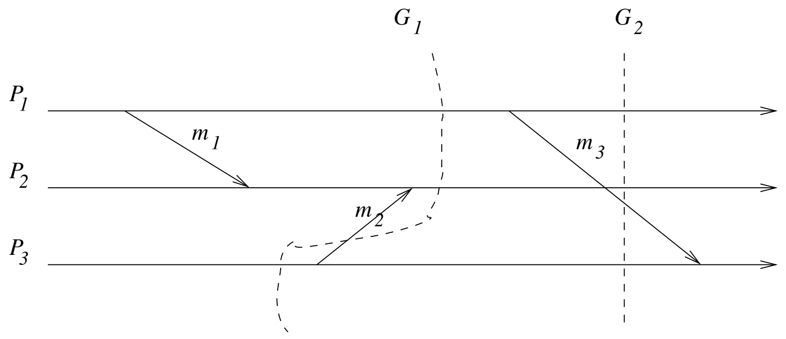

Figure 9.1: Consistent and inconsistent cuts

Figure 9.1 depicts a distributed computation. The dashed lines labeled G_1 and G_2 represent global states that consist of local states at P_1, P_2, and P_3, where G_1 and G_2 intersect the processes. Because a global state can be visualized in such a figure as a *cut* across the computation, the term, "cut" is used interchangeably with "global state." The cut G_1 in this computation is not consistent because it records the message m_2 as having been received but not sent. This is clearly impossible. The cut G_2 is consistent. The message m_3 in this cut has been sent but not yet received. Thus it is a part of the channel from process P_1 to P_3.

Formally, in a state based model of a computation $(S, \rightarrow)$, let G be a set of local states with exactly one local state from each process. Let $G[i]$ be the local state from P_i. Then, G is called a **consistent global state** iff

$$\forall i, j : G[i] || G[j]$$

In an event-based model of a computation $(E, \rightarrow)$, with total order $\prec$ on events in a single process, we define a *cut* as any subset $F \subseteq E$ such that

$$f \in F \wedge e \prec f \Rightarrow e \in F.$$

We define a **consistent cut**, or a **consistent global state**, as any subset $F \subseteq E$ such that

$$f \in F \wedge e \to f \Rightarrow e \in F.$$

Fig. 9.2 shows the trace of instructions executed for a distributed computation. Fig. 9.3 and Fig. 9.4 show the event based and the state based models for the computation.

Proc. 1	Proc. 2
1: local event (a)	1: local event (e)
2: send msg (b)	2: receive msg (f)
3: local event (c)	3: local event (g)

Figure 9.2: Pseudocode of instructions

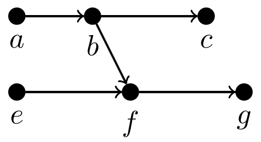

Figure 9.3: Event Based Model

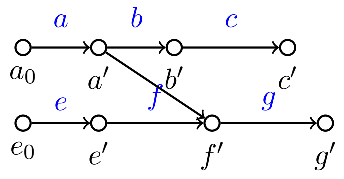

Figure 9.4: State Based Model

Fig. 9.5 and Fig. 9.6 show the set of consistent global states in the event based and the state based model.

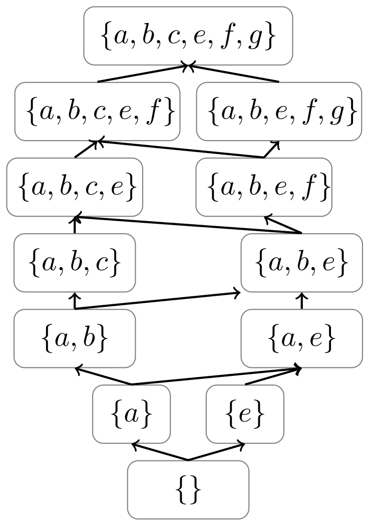

Figure 9.5: Lattice of Consistent Global States in event based model

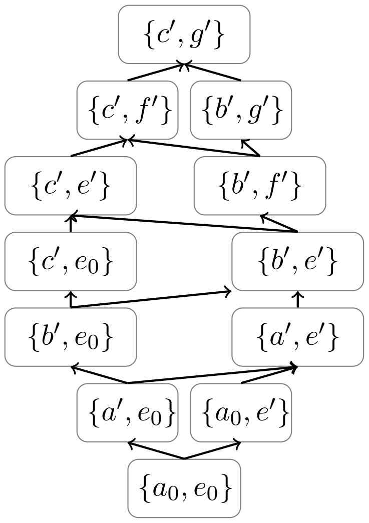

Figure 9.6: Lattice of Consistent Global States in the state based model

9.2 Chandy and Lamport's Global Snapshot Algorithm

In this section, we describe an algorithm to take a global snapshot (or a consistent cut) of a distributed system. Our example of the distributed database in the previous section illustrates the importance of recording only the consistent cuts. The computation of the snapshot is initiated by one or more processes. We assume that all channels are unidirectional and satisfy the FIFO property. Assuming that channels are unidirectional is not restrictive because a bidirectional channel can simply be modeled by using two unidirectional channels. The assumption that channels are FIFO is essential to the correctness of the algorithm as explained later.

The algorithm is shown in Figure 9.8. We associate with each process a variable called *color* that is either white or red. Intuitively, the computed global snapshot corresponds to the state of the system just before the processes turn red. All processes are initially white. After recording the local state, a process turns red. Thus the state of a local process is simply the state just before it turned red.

There are two difficulties in the design of rules for changing the color for the global snapshot algorithm: (1) we need to ensure that the recorded local states are mutually concurrent, and (2) we also need a mechanism to capture the state of the channels. To address these difficulties, the algorithm relies on a special message called a *marker*. Once a process turns red, it is required to send a marker along all its outgoing channels before it sends out any message. A process is required to turn red on receiving a marker if it has not already done so. Since channels are FIFO, the above mentioned rule guarantees that no white process ever receives a message sent by a red process. This in turn guarantees that local states are mutually concurrent.

Now let us turn our attention to the problem of computing states of the channels. Figure 9.7 shows that messages in the presence of colors can be of four types:

1. *ww messages*: These are the messages sent by a white process to a white process. These messages correspond to the messages sent and received before the global snapshot.

2. *rr messages*: These are the messages sent by a red process to a red process. These messages correspond to the messages sent and received after the global snapshot.

3. *rw messages*: These are the messages sent by a red process received by a white process. In the figure, they cross the global snapshot in the backward direction. The presence of any such message makes

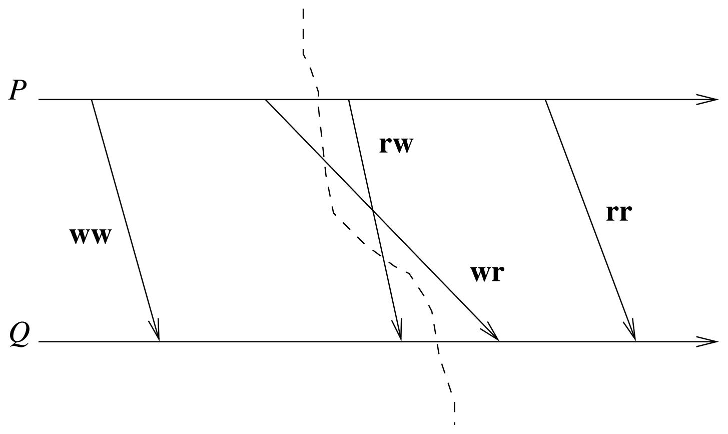

Figure 9.7: Classification of messages

the global snapshot inconsistent. The reader should verify that such messages are not possible if a *marker* is used.

4. *wr messages*: These are the messages sent by a white process received by a red process. These messages cross the global snapshot in the forward direction and form the state of the channel in the global snapshot because they are in transit when the snapshot is taken.

To record the state of the channel, P_j starts recording all messages it receives from P_i after turning red. Since P_i sends a marker to P_j on turning red, the arrival of the marker at P_j from P_i indicates that there will not be any further white messages from P_i sent to P_j. It can, therefore, stop recording messages once it has received the marker.

The program shown in Figure 9.8 uses `chan[k]` to record the state of the kth incoming channel and `closed[k]` to stop recording messages along that channel. In the program, we say that P_j is a *neighbor* of P_i if there is a channel from P_i to P_j.

The method `turn_red` turns the process red, records the local state, and sends the *marker* message on all outgoing channels. On receiving a marker message, if the process is white, it turns red by invoking `turn_red`. It also sets `closed[src]` to true because there cannot be any message of type *wr* in that channel after the marker is received. An application message is recorded if the receiving process is red and the incoming channel is not closed (i.e., the sender is white).

In the algorithm, whenever a process turns red, it notifies all its neighbors. On receiving any such notification, that neighbor is required to update its own color. This may result in additional messages if that neighbor turned red. The net result is that if one process turns red, all processes that can be reached directly or indirectly from that process also turn red.

The Chandy-Lamport algorithm requires that a marker be sent along all channels. Thus it has an overhead of e messages, where e is the number of unidirectional channels in the system. We have not discussed the overhead required to combine local snapshots into a global snapshot. A simple method would be for all processes to send their local snapshots to a predetermined process, say, P_0.

9.3 Global Snapshots for non-FIFO Channels

We now describe an algorithm due to Mattern that works even if channels are not FIFO. We cannot rely on the marker any more to distinguish between white and red messages. Therefore, we include the color in all the outgoing messages for any process besides sending the marker. Further, even after P_i gets a red

```
$P_i$::
    var
        $color$: $\{white, red\}$ initially $white$;
        // assume $k$ incoming channels
        $chan$: array[$1..k$] of queues of messages initially $null$;
        $closed$: array[$1..k$] of boolean initially $false$;

    $turn\_red()$ enabled if $(color = white)$:
        save_local_state;
        $color := red$;
        send (marker) to all neighbors;

    Upon $receive(marker)$ on incoming channel $j$:
        if (color = white) then
            turn_red();
        $closed[j] := true$;

    Upon $receive(program\_message)$ on incoming channel $j$:
        if $(color = red) \wedge \neg closed[j]$ then // append the message
            $chan[j].append(program\_message)$;
```

Figure 9.8: Chandy and Lamport's snapshot algorithm

message from P_j or the marker, it cannot be sure that it will not receive a white message on that channel. A white message may arrive later than a red message due to the overtaking of messages. To solve this problem we include in the marker the total number of white messages sent by that process along that channel. The receiver keeps track of the total number of white messages received and knows that all white messages have been received when this count equals the count included in the marker. We leave the details of the algorithm to the reader as an exercise.

9.4 Channel Recording by the Sender

Chandy and Lamport's algorithm requires the receiver to record the state of the channel. Since messages in real channels may get lost, it may be advantageous for senders to record the state of the channel. We will assume that control messages can be sent over unidirectional channels even in the reverse direction.

The mechanism to ensure that we do not record inconsistent global state is based on the coloring mechanism discussed earlier. A process sends white messages before it has recorded its local state and red messages after it has recorded the local state. By ensuring that a white process turns red before accepting a red message, we are guaranteed that there are no rw messages and therefore we will record only a consistent global snapshot.

Now let us turn our attention to recording the state of channels. We assume that the sender records all the messages that it sends out on any outgoing channel before it turned red. Whenever a process turns red, it sends a marker message on all its incoming channels (in the reverse direction) indicating the messages it has received on that channel so far. The sender can now compute the state of the channel by removing from its buffer all messages that have been received according to the marker.

In this scheme, the sender may end up storing a large number of messages before the marker arrives. Assuming that all control messages (marker and acknowledgment messages) follow FIFO ordering, we can reduce the storage burden at the sender by requiring the receiver to send acknowledgments. When the sender receives an acknowledgment and has not received the marker, it can delete the message from the storage. To identify each message uniquely, we can use sequence numbers. Thus, the algorithm can be summarized by the following rules.

1. Every process is white before recording its state and red after recording its state. A white process sends white messages and a red process sends a red message.

2. A white process turns red before accepting a red message or a marker.

3. On turning red, a process sends markers on all incoming channels in the reverse direction.

4. A white process acknowledges a white message.

5. A white process records any message sent. On receiving an acknowledgment, the corresponding message is removed from the record.

Note that this algorithm also does not require channels for application messages to be FIFO. If channels are known to be FIFO then the receiver only needs to record the sequence number of the last message it received before turning red. The algorithm does require the ability to send control messages in the reverse direction for any application channel. Furthermore, it requires control messages to follow FIFO order. (Why?) If the underlying network does not support FIFO, then sequence numbers can be used to ensure FIFO ordering of messages.

9.5 Problems

9.1. Show that if G and H are consistent cuts of a distributed computation $(E, \rightarrow)$, then so are $G \cup H$ and $G \cap H$.

9.2. The global snapshot algorithms discussed in this chapter do not freeze the underlying computation. In some applications it may be okay for the underlying application to be frozen while the snapshot algorithm is in progress. How can the snapshot algorithm be simplified if this is the case? Give an algorithm for global snapshot computation and its Java implementation.

9.3. Extend the Java implementation of Chandy and Lamport's algorithm to allow repeated computation of global snapshots.

9.4. The original algorithm proposed by Chandy and Lamport does not require FIFO but a condition weaker than that. Specify the condition formally.

9.5. How can you use Lamport's logical clock to compute a consistent global snapshot?

9.6. Give Java implementation of global snapshot algorithm when channels are not FIFO.

9.7. Extend Chandy and Lamport's algorithm to compute a *transitless global state.* A consistent global state is transitless if there are no messages in any channel in that global state. Note that a process may have to record its local state multiple times until the recorded local state can be part of a transitless global state. Give Java implementation of your algorithm.

9.8. Give an example of a distributed computation in the interleaving model (with the events of the superimposed global snapshot algorithm) in which the recorded global snapshot does not occur in the computation.

9.9. How will you use snapshot algorithms to detect that the application has reached a deadlock state?

9.6 Bibliographic Remarks

Chandy and Lamport [CL85] were the first to give an algorithm for computation of a meaningful global snapshot (a colorful description of this algorithm is given by Dijkstra [Dij85]). Spezialetti and Kearns have given efficient algorithms to disseminate a global snapshot to processes initiating the snapshot computation [SK86]. Bouge [Bou87] has given an efficient algorithm for repeated computation of snapshots for synchronous computations. In the absence of the FIFO assumption, as shown by Taylor [Tay89], any algorithm for a snapshot is either inhibitory (that is, it may delay actions of the underlying application) or requires piggybacking of control information on basic messages. Lai and Yang [LY87] and Mattern [Mat93] have given snapshot algorithms that require only the piggybacking of control information. Helary [Hel89] has proposed an inhibitory snapshot algorithm.