GLOBAL STATE

On many occasions, it is useful to know the global state in which a distributed system is currently residing. The **global state** of a distributed system consists of the local state of each process, together with the messages that are currently in transit, that is, that have been sent but not delivered. What exactly the local state

of a process is depends on what we are interested in (Helary, 1989). In the case of a distributed database system, it may consist of only those records that form part of the database and exclude temporary records used for computations. In our example of tracing-based garbage collection as discussed in the previous chapter, the local state may consist of variables representing markings for those proxies, skeletons, and objects that are contained in the address space of a process.

Knowing the global state of a distributed system may be useful for many reasons. For example, when it is known that local computations have stopped and that there are no more messages in transit, the system has obviously entered a state in which no more progress can be made. By analyzing such a global state, it may be concluded that we are either dealing with a deadlock (see, for example, Bracha and Toueg, 1987), or that a distributed computation has correctly terminated. An example of how such an analysis can actually be done is discussed below.

A simple, straightforward way for recording the global state of a distributed system was proposed by Chandy and Lamport (1985) who introduced the notion of a **distributed snapshot**. A distributed snapshot reflects a state in which the distributed system might have been. An important property is that such a snapshot reflects a consistent global state. In particular, this means that if we have recorded that a process P has received a message from another process Q, then we should also have recorded that process Q had actually sent that message. Otherwise, a snapshot will contain the recording of messages that have been received but never sent, which is obviously not what we want. The reverse condition (Q has sent a message that P has not yet received) is allowed, however.

The notion of a global state can be graphically represented by what is called a **cut**, as shown in Fig. 5-9. In Fig. 5-9(a), a consistent cut is shown by means of the dashed line crossing the time axis of the three processes P_1, P_2, and P_3. The cut represents the last event that has been recorded for each process. In this case, it can be readily verified that all recorded message receipts have a corresponding recorded send event. In contrast, Fig. 5-9(b) shows an inconsistent cut. The receipt of message *m2* by process *P3* has been recorded, but the snapshot contains no corresponding send event.

To simplify the explanation of the algorithm for taking a distributed snapshot, we assume that the distributed system can be represented as a collection of processes connected to each other through unidirectional point-to-point communication channels. For example, processes may first set up TCP connections before any further communication takes place.

Any process may initiate the algorithm. The initiating process, say P, starts by recording its own local state. Then, it sends a marker along each of its outgoing channels, indicating that the receiver should participate in recording the global state.

When a process Q receives a marker through an incoming channel C, its action depends on whether or not it has already saved its local state. If it has not

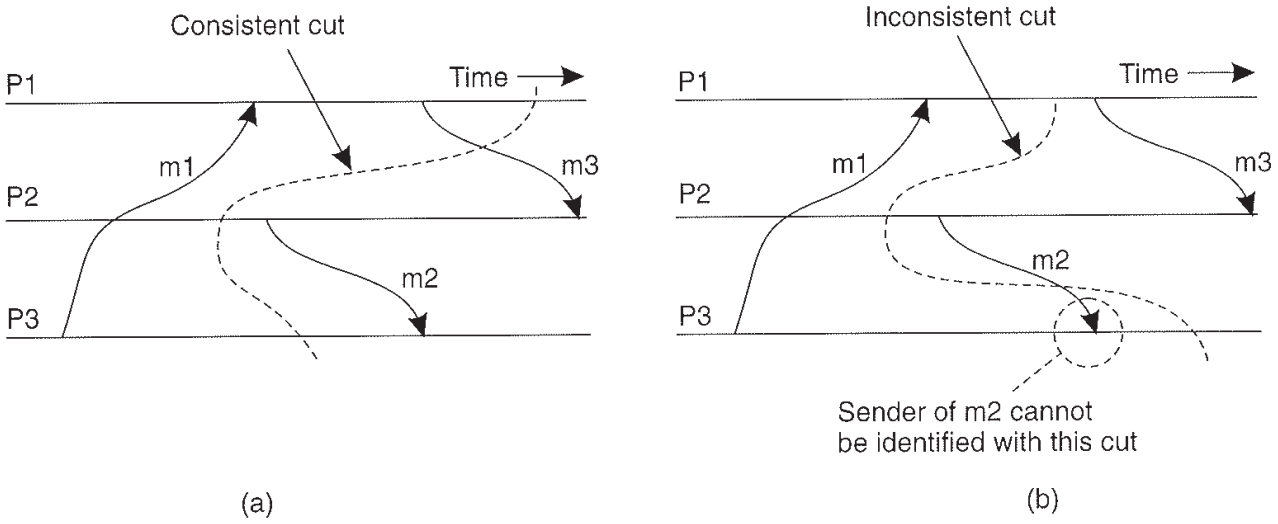

Figure 5-9. (a) A consistent cut. (b) An inconsistent cut.

already done so, it first records its local state and also sends a marker along each of its own outgoing channels. If Q had already recorded its state, the marker on channel C is an indicator that Q should record the state of the channel. This state is formed by the sequence of messages that have been received by Q since the last time Q recorded its own local state, and before it received the marker. Recording this state is shown in Fig. 5-10.

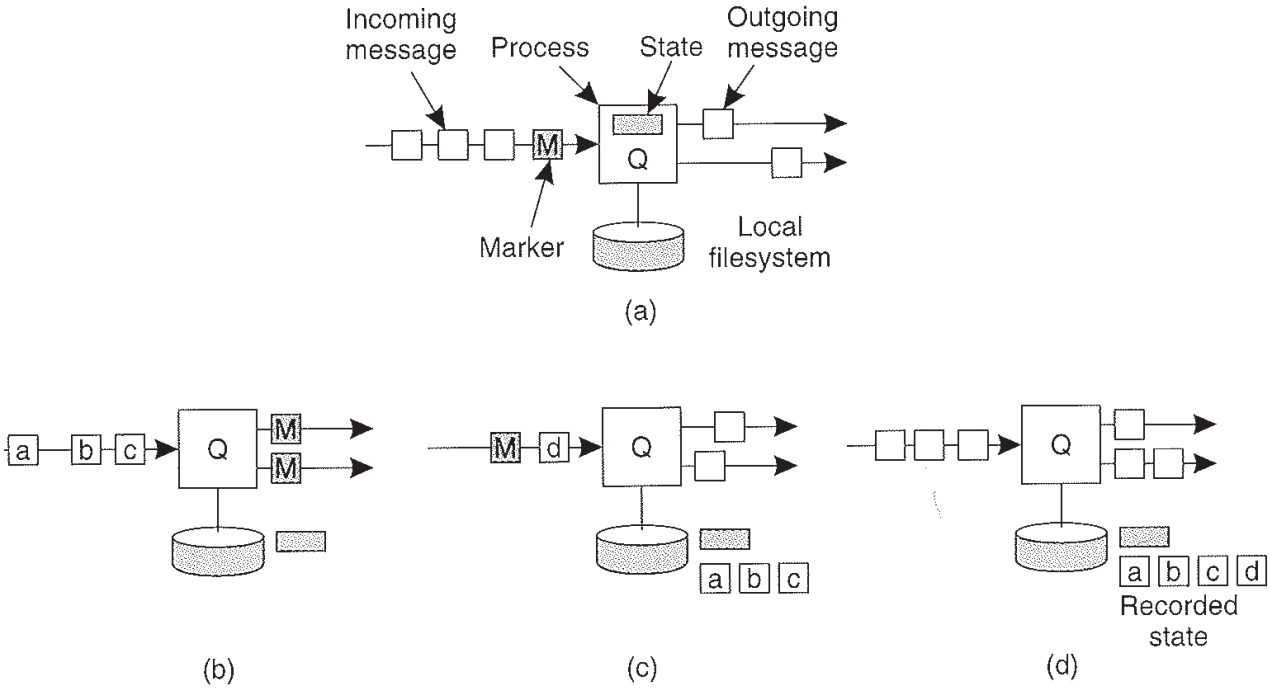

Figure 5-10. (a) Organization of a process and channels for a distributed snapshot. (b) Process Q receives a marker for the first time and records its local state. (c) Q records all incoming messages. (d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel.

A process is said to have finished its part of the algorithm when it has received a marker along each of its incoming channels, and processed each one. At that point, its recorded local state, as well as the state it recorded for each incoming channel, can be collected and sent, for example, to the process that initiated the snapshot. The latter can then subsequently analyze the current state. Note that, meanwhile, the distributed system as a whole can continue to run normally.

It should be noted that because any process can initiate the algorithm, the construction of several snapshots may be in progress at the same time. For this reason, a marker is tagged with the identifier (and possibly also a version number), of the process that initiated the snapshot. Only after a process has received that marker through each of its incoming channels, can it finish its part in the construction of the marker's associated snapshot.

Example: Termination Detection

As an application of taking a snapshot, consider detecting the termination of a distributed computation. If a process Q receives the marker requesting a snapshot for the first time, it considers the process that sent that marker as its predecessor. When Q completes its part of the snapshot, it sends its predecessor a *DONE* message. By recursion, when the initiator of the distributed snapshot has received a *DONE* message from all its successors, it knows that the snapshot has been completely taken.

However, a snapshot may show a global state in which messages are still in transit. In particular, suppose a process records that it had received messages along one of its incoming channels between the point where it had recorded its local state, and the point where it received the marker through that channel. Then, clearly, we cannot conclude that the distributed computation is completed, for those messages may have generated other messages that are not part of the snapshot.

What is needed is a snapshot in which all channels are empty. The following is a simple modification to the algorithm described above. When a process Q finishes its part of the snapshot, it either returns a *DONE* message to its predecessor, or a *CONTINUE* message. A *DONE* message is returned only when the following two conditions are met:

1. All of Q's successors have returned a *DONE* message.
2. Q has not received any message between the point it recorded its state, and the point it had received the marker along each of its incoming channels.

In all other cases Q sends a *CONTINUE* message to its predecessor.

Eventually, the original initiator of the snapshot, say process P, will either receive a *CONTINUE* message, or only *DONE* messages from its successors. When only *DONE* messages are received, it is known that no regular messages are in transit, and thus the computation has terminated. Otherwise, process P initiates another snapshot, and continues to do so until only *DONE* messages are eventually returned.

Numerous other solutions to termination detection as discussed in this section have been developed. See (Andrews, 2000; and Singhal and Shivaratri, 1994) for further examples and references. An overview and comparison of different solutions can also be found in (Mattern, 1987; and Raynal, 1988).